University of
Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master in Computer Science	Spring semester, 2011 Open / Restricted access
Writer: Yu Xiao	…………………………………………. (Writer's signature)
Faculty supervisor: Chunming Rong, Tomasz Wiktor Włodarczyk External supervisor(s):	
Title of thesis: Semantic Query Reasoning in Distributed Environment	
Credits (ECTS): 30	
Key words: Semantic Web, Ontology, cloud computing, data materialization, query rewriting	Pages: ……54………… + enclosure: ………… Stavanger, 14/06/2011 Date/year

Abstract

Semantic Web aims to elevate simple data in WWW to semantic layer, so that knowledge, processed by machine, can be shared more easily. Ontology is one of the key technologies to realize Semantic Web. Semantic reasoning is an important step in Semantic technology. For Ontology developers, semantic reasoning finds out collisions in Ontology definition, and optimizes it; for Ontology users, semantic reasoning retrieves implicit knowledge from known knowledge.

The main research of this thesis is reasoning of semantic data querying in distributed environment, which tries to get correct results of semantic data querying, given Ontology definition and data. This research studied two methods: data materialization and query rewriting. Using Amazon cloud computing service and LUBM, we compared these two methods, and have concluded that when size of data to be queried scales up, query rewriting is more feasible than data materialization. Also, based on the conclusion, we developed an application, which manages and queries semantic data in a distributed environment. This application can be used as a prototype of similar applications, and a tool for other Semantic Web researches as well.

Contents

Acknowledgements

As about to finish, I would like to give many thanks to those who have helped me during my study and the period when I wrote the thesis. First of all, I'd like to show sincere gratitude to my supervisor Professor Chunming Rong, who gave me a lot of help and useful advices on my research and personal life as well. Secondly, I’d like to thank PhD student Tomasz Wiktor Włodarczyk who provided me the topic of the thesis, for his consistant help during our research. I ’d like to thank Yi Han, together with whom I did the research of the thesis. It’s a pleasure working with him. Also I’d like to thank all my family and friends, especially Alice Liao. Without you, it would be much harder for me to finish my study and the thesis. Last but not the least, I’d like to thank the University of Stanvanger and our Faculty of Science and Technology. You gave me the opportunity to study in such a great country, and these two years will definitely be precious in my memory.

Chapter 1 Introduction

In the domain of computer science, Ontology is a formal expression to some domain knowledge. It contains concepts and relationship between them [1], which can be used to describe objects in a domain and reason based on the description. As a way to express knowledge, Ontology has been widely used in areas such as Artificial Intelligence, Semantic Web, System Engineering, Software Engineering, Biomedical Informatics, Enterprise Bookmark, Information System Structure, etc. [2]

To formalize knowledge expression, and make knowledge discovery, retrieval, sharing easier and more effective, nowadays, domain Ontologies have been built in many industries (such as energe industry[3][4], medical industry[5][6], Biology [7][8], Astronomy[9]). Some of the famous ones are: COSMO[10], a fundamental Ontology used to define all knowledge concept primitives; BioPAX[11], used to integrate and exchange Biological pathway data; Gene Ontology[12], used to formalize expression of Gene information; ISO 15926[13], used to integrate date during oil&gas development lifecircle. Ontologies like these are often comlex, and once actual data is input into them, the size of them will grow so large that managing and processing them locally will be no longer applicable [14]. In practice, Ontology definition and Ontology data are usually separated apart. Ontology definition is published and quoted by different user parties, while Ontology data may have diverse formats and storage schemes according to its user. Even in a single user party, Ontology data is often distributedly stored and processed because of its size. In our research, we will focus on process Ontology data in a distributed environment.

Querying an Ontology's classes and instances is a common service once an Ontology has been created. Different from traditional querying such as relational database querying, querying an Ontology involves more semantic and logic operations [15]. We have to consider whether the query results conform to the Ontology definition, whether all logical results have been retrieved, whether unreasonable results are also included. All our efficiency study comes after the correctness of semantic querying.

It's inefficient when Ontology data is stored together with Ontology definition using the same format such as OWL. In practice, people usually define Ontology using language like OWL, while store data in data files or database, and then establish mapping between definition and actuall data.

In this case, we can access data efficiently, and on the other hand, gain the expressive power of Ontology.

Our research use experimental approach to compare the excutive efficiency of the two semantic querying methods in a distributed environment, and develop an application for further research according to the comparison. The second chapter of this thesis will introduce some background knowledge, on which our research is based; the third chapter detailedly describes what we are going to research and how we do it; the fourth chapter gives the tests and comparisons we make, analysize test results and then make conclusions; the fifth chapter introduces the development of the application; the sixth chapter is to test the application and make necessary improvement; the last chapter concludes the whole thesis.

Chapter 2 Background

2.1 Sematic Web

Semantic Web is an extensiont to WWW (World Wide Web). It's a combination of a set of techniques, which aims to make computers understand the meaning, or say, "semantic" of data on Internet [16]. WWW we have now is document-oriented, while Semantic Web is data-oriented. In Semantic Web, information is meaningful to computers, so that it can be processed and intergrated. In other words, Semantic Web is a smart web, which understands human language, and makes communication between human and computer as easy as communication between humans.

Berners Lee put forward Semantic Web architecture in 2000. The architecture has 7 layers, as shown in Figure 2.1. [17]

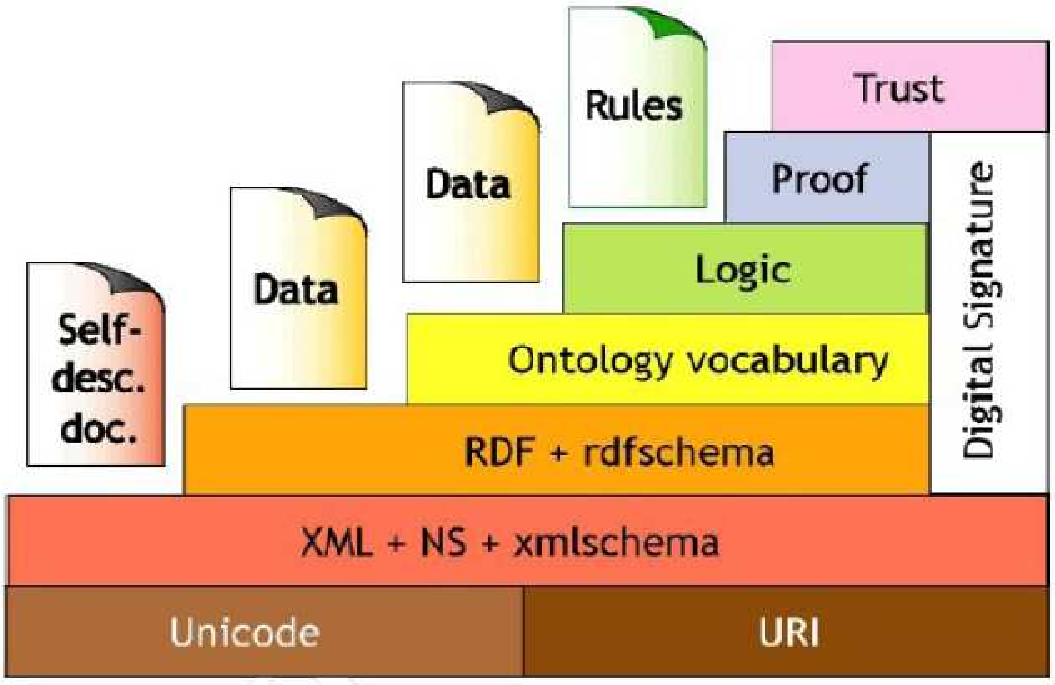


Figure 2.1 Semantic Web Architecture (Tim Berners-Lee, 2000)

Layer 1: Unicode and URI. The first layer is the basis of the whole architecture. Unicode encodes all resources; URI allocates each resource a unique name. Layer 2: XML + NS + xmlschema. NS (Name Space), decided by URI, avoids naming collision between different applications. XMLSchema [18] is a replacement of DTD. It adopts XML grammar, but is more flexible than DTD, and provides more data types, so it's a better data check mechanism for XML documents. This layer expresses data content and structure in terms of grammar. It separates information content from data format and structure by using standard language. Layer 3: RDF + rdfschema. RDF [19] is a WWW information description language, which aims at building up a framework to let multiple data standards coexist. While we regard XML as a standardized metadata grammar criterion, RDF is a standardized metadata semantic description criterion. Rdfschema defines resource description vocabulary, which is understandable to machine. Layer 4: Ontology vocabulary. Based on RDF(S), this layer extends vocabulary and defines concepts and relations between them. It is used to describe domain knowledge, resources and relations between resources. Layer 5 to Layer 7: Logic, Proof, Trust. Logic is in charge of axiom and reasoning rules. W can validate logic through logic reasoning to resources and relations between them once it has been established. Trust can be established by exchanging proof and digital signature, which verifies reliability of Semantic Web output, and whether the output conforms to what users have required.

There are three key techniques to realize Semantic Web: XML, RDF and Ontology. At present, the main focus is on RDF and Ontology.

2.2 Ontology

In recent years, Ontology has been a hot key word in Information Technology. Originally ontology is a branch of philosophy, which studies essence of existence of objective things. In Computer Science, Ontology is part of conten theories in the domain of Artificial Intelligence, which studies object classification, object attribute and relations between objects. It provides terminology for domain knowledge description. Ontology plays an important role in areas such as information exchanging, system integration, knowledge-based software development, etc.

Normally, an Ontology describes following aspects:

（1） Instances: basic and fundamental element in Ontology. Instances in an Ontology can be actual objects, and abstract objects like text and numbers as well. Instances are not necessary to an Ontology.

（2） Classes: classes are abstraction of objects. An instance of a class can be an individual, or another class.

（3） Attributes: objects in an Ontology can be described by assigning attribute values to them. An attribute contains at least one name and one value, which stores specific information of an object.

（4） Relations: an important contribution of attributes is to describe relations between two objects. Usually a relation is an attribute which has another object in the Ontology as its value. In general, the set of relations describes the whole semantic in a domain.

（5） Events: events are objects about time, or instantiated object resources.

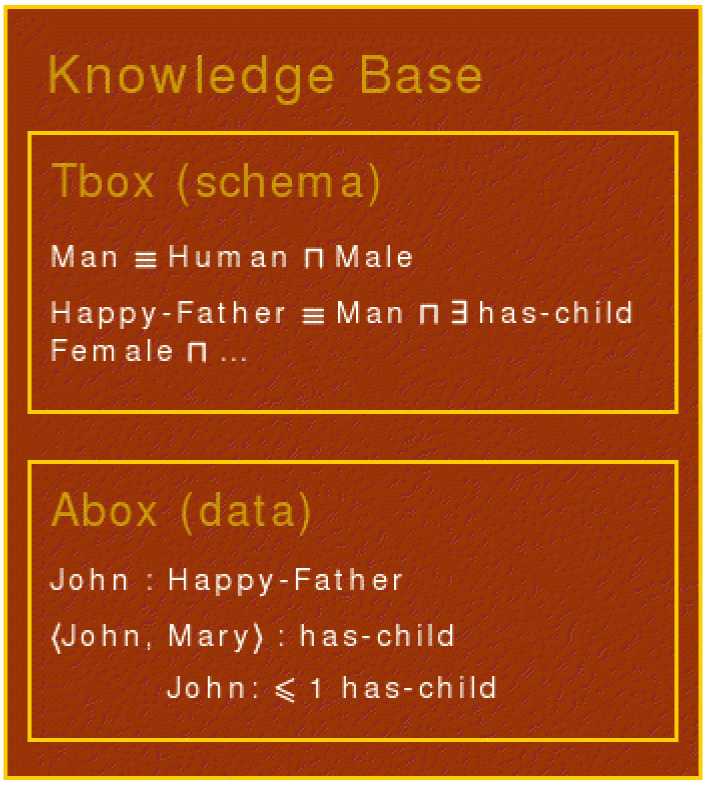
The objective of Ontology is to capture domain knowledge, provide shared understanding to domain knowledge, make sure that only one set of vocabulary is used, and clearly define terms and relationships between terms on different layers. Generally, creating Ontology makes kownledge sharable and reusable to some extent, and improves communication, interoperability, reliability of a system.

OWL [20] (Web Ontology Language) is developed by W3C, which is used to describe semantic. There are three subsets of OWL: Lite, DL and Full. OWL Lite is the least expressive one, a subset of OWL DL. It assures efficient reasoning by reducing axiom constraints in OWL DL. OWL DL (Description Logic) contains all elements of OWL, but is restrictedly used. OWL DL provides reasoning functions in description logic, which is formalized basis of OWL. OWL Full contains all elements of OWL with no restriction. It extends RDFS to a complete Ontology language, and is suitable for RDFS users who need no computational guarantees while the best expressive power with no limit.

2.3 Ontology Reasoning

The task of reasoning is to mine implicit knowledge out of existing knowledge. To Ontology developers, reasoning checks collisions in an Ontology, optimizes Ontology expression and integrates different Ontologies. To Ontology users, reasoning obtains specific knowledge set in an Ontology and solves problems using knowledge in an Ontology. [21]

Ontology reasoner is theoretically based on Description Logic. Description Logic is formalization of object-based knowledge representation. It describes knowledge according to binary relation between concepts. A DL knowledge base consists of a TBox and an ABox (shown in Figure 2.2). A TBox contains abstract logic relations between concepts. For example, in Figure 2.2, “Man” is the intersection of “Human” and “Male”. An ABox contains actual concept objects and decriptions to logic relations between them. For example, in Figure 2.2, “John” is asserted as a “Happy-Father”, and “John” “has-child” “Mary”. [22]


Knowledge Base

Tbox (schema)

$\text{Man} \equiv \text{Human} \sqcap \text{Male}$

$\text{Happy-Father} \equiv \text{Man} \sqcap \exists \text{has-child Female} \sqcap \ldots$

Abox (data)

John : Happy-Father

$\langle \text{John, Mary} \rangle$: has-child

John: $\leqslant 1$ has-child

Figure 2.2 Description Logic Composition

At present, some research groups have developed some practical reasoners, such as Pellet, Racer, Fact, etc. Jena, a Java framework used to construct Semantic Web applications, also provides built-in reasoners.

2.4 Ontology Query Language

We query Ontology in order to obtain knowledge. Some popular query languages are: DQL, OWL QL, nRQL, RDQL and SPARQL. OWL QL is developed based on DQL, by KSL lab of Stanford University. nRQL is for description logic reasoner: Racer, which is an extension of RQL. RDQL is a built-in RDF query language of Jena. It doesn't contain a reasoning mechanism, and has to combine an external reasoner. SPARQL was recommented as a standard candidate in 2006. Based on RDQL grammar, it can be used for any data source which can be mapped as RDF. [23]

2.5 Cloud Computing

Cloud computing is a concept which combines traditional technologies such as grid computing, distributed computing, parallel computing, utility computing, distributed storage, virtualization, load balancing, etc [24]. It aims to integrate multiple low-cost computing entities into a powerful computing system through network, and distribute the computing capacity to end users with the help of business models like SaaS[25] (Software-as-a-service), PaaS[26] (Platform-as-a-Service), IaaS (Infrastructure as a Service), MSP(Managing Successful Program), etc. The ambition of cloud computing is to enhance processing ability of the "cloud", cut down user terminals' load, and eventually simplify user terminals as a pure I/O device, which share the powerful computing and processing ability of the "cloud".

Figure 2.3 Cloud Computing Concept

Brought up by Google, cloud computing has entered practical usage stage after studied by thousands of researchers. Amazon released EC2 and S3 to provide enterprises computing and storage service. Yahoo! also has its cloud computing service, whose PNUTS is a large-scale paralelle, distributed data platform [27]. Yahoo! is one of the earliest supplier who provides open source cloud computing servers. Google is the largest cloud computing user. Google's searching engine is built upon over 1 million servers, acrossing 200 locations, and the numbers are still growing. Google Maps, Google Earth, Gmail, Google Docs also use the same infrastructure. Now Google has allowed third-party to utilize its cloud computing by running paralelle applications through Google App Engine. In November 2007, IBM launched its "Blue Cloud" platform, which was said to be "rule-changing" [28]. It contains a series of automated, self-managing and self-repairing virtulized cloud computing software, to let global applications visit its distributed server pool. Microsoft also followed the trend. It launched "Windows Azure" [29] operating system in October 2008. It's intended to build up a new cloud computing platform on Internet, and extend Windows from PC to "Azure".

2.6 Motivation and Existing Research

The focus of our research is on query reasoning of semantic data in a distributed environment, whose application background is: in a specific domain, assume that domain knowledge base has been established by means of Ontology. The users of the knowledge base are enterprises or researchers, who may need to process the same dataset based on the domain knowledge in different locations, which is common and typical. Except for semantic data querying, we will also discuss semantic data management such as inserting and deleting.

There are many problems in Artificial Intelligence and Semantic Web area which are worth concerning. Query reasoning aims to achieving correctness (or accurateness) when obtaining knowledge data. We have similar examples around us. For example, when web searching, after we input keywords, we usually can't get satisfying and accurate results. Mostly we need more processing to the mass of results. One of the most important reasons is that the data on Internet is not organized according to users' domain knowledge. Similarly, when processing semantic data, people do not strictly respect the definition of domain knowledge. For instance, assume that a semantic about family contains relations as below: father and grandfather. Grandfather is defined as the father of father..In a dataset, we have semantic data: B is father of A, D is father of C, E is father of B, and F is grandfather of C. If we search for all instances of grandfather, normally we get asserted F. According to our semantic, since B is father of A, and E is father of B, E should be grandfather of A, so the corrct result should be E and F. The purpose of query reasoning is to find out all correct results according to semantic definition.

There are two ways to achieve semantic querying which are frequently used: one is materialization, the other is query rewriting.

Materialization refers to loading Ontology model, adding original data, running reasoner and then getting reasoned model. A reasoned model contains information which is implicit in original model, while unveiled by reasoner. Querying the reasoned model, we shall get correct results. (Shown in Figure 2.4)

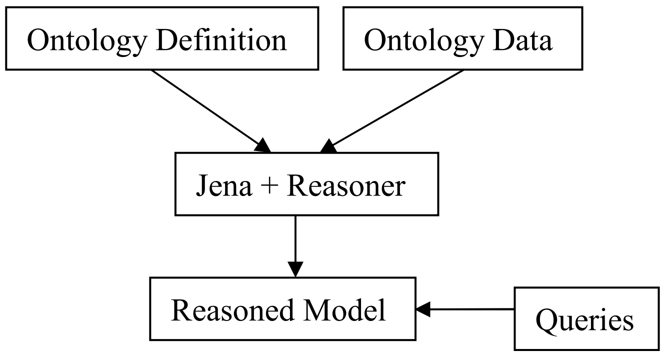


Figure 2.4 Materialization

There are a lot of researches and applications about materialization. By studying the mode of user querying, Naveen Ashish materialized part of web data to improve query efficiency [30]. Joseph Fong built up a semantic metadata for heterogeneous data in different models, and allowed users to decide how to materialize data according to their needs [31].

Query rewriting refers to rewriting queries according to Ontology semantic, and then querying original Ontology model using new series of queries. To rewrite queries, we use a tool called Requiem in our research. Requiem is a prototype implementation to a query rewriting algorithm. Given Ontology definition (OWL document) and a query that needs to be rewritten as input, Requiem outputs a series of new queries reasoned according to the Ontology. (Shown in Figure 2.5)

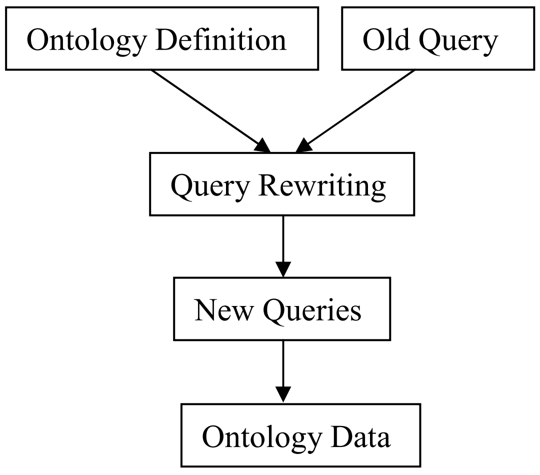


Figure 2.5 Query Rewriting

Maria Esther Vidal and her collegues brought up a 3-layer model, which contains Ontology layer, data layer and physical layer. Going through Ontology and data layer, queries are rewritten and get to physical layer. Their approach has improved query accurateness and efficiency [32]. Alexandre Riazanov studied about semantic querying to relational database. They used a reasoner to rewrite a query into several SQL queries, in order to get high expressiveness at a reasonable price

[33]. Roger Castillo reduced “Join” operations during querying process by “materializing” (rewriting) queries and using storage strategy suitable for querying, which improved querying efficiency [34].

To test and compare these two approaches, we make use of a data benchmark provided by Semantic Web and Agent Technology Lab of Lehigh University, LUBM. The purpose of LUBM is to help researchers make standard and systematic measurements to Semantic Web applications [35]. It contains an Ontology about universities, regenerable and repeatable artificial data and a set of queries (Appendix 1). The class structure of the Ontology is shown below:

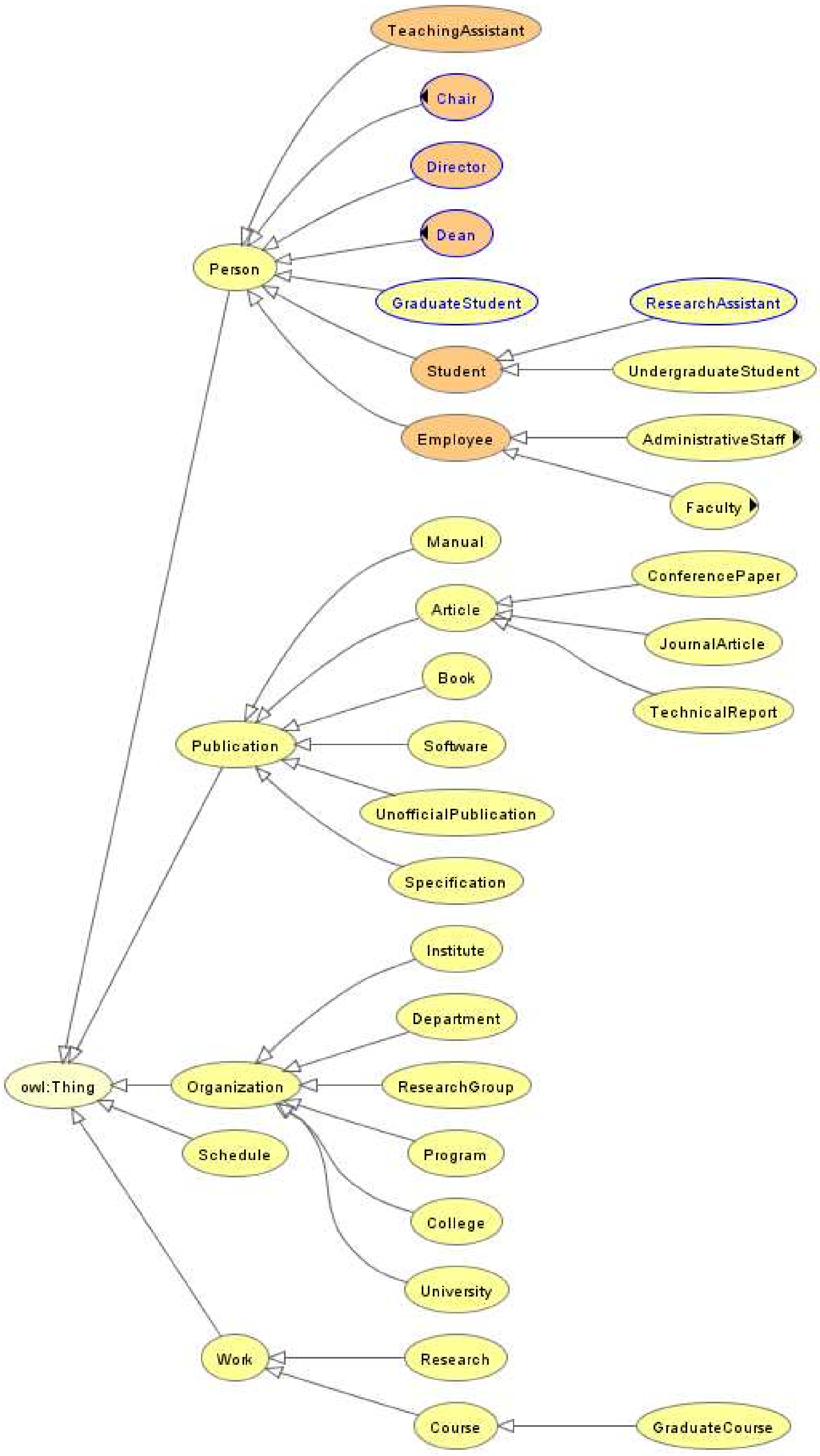


Figure 2.6 LUBM Ontology Classes

Chapter 3 Experiments and Tests

In this chapter, we will take a series of experiments and tests according to our plan.

3.1 Text Data Querying based on Hadoop

In order to test in a distributed environment, we make use of a Web service provided by Amazon: AWS. AWS is a set of remote computing services, which gives people a cloud computing platform. The two key services of AWS are Amazon EC2 and Amazon S3. EC2 leases virtual computers (which are call “instances” by Amazon) to users, on which people can run programs. Amazon S3 is an online storage service, which provides users almost limitless storage ability. AWS services are not free though.

It’s not enough if we have only AWS, we need also a distributed software framework for those virtual computers. Here we use Hadoop [36]. Hadoop is a fundamental distributed system framework, developed by Apache. Without knowing underlying details, users can develop distributed applications and make use of the powerful computing and storage ability of clusters [37]. Hadoop originates from three important technologies of Google: GFS [38], MapReduce [39] and BigTable [40]. GFS（Google File System） is a distributed file system. Hiding its underlying details such as load balancing, redundancy management, GFS provides a unified file system API to upper layer. Google found out that most distributed processes could be abstracted as MapReduce operations. Map breaks down input into intermediate Key/Value pairs, and Reduce combines these Key/Values into output. Map and Reduce functions are given by programmer, and underlying infrastructure distributes Map and Reduce operations to the cluster, and then stores the output on GFS. BigTable is a distributed database. It’s not a relational database. Like its name, it’s a large table, used to store structural data. In the beginning, Hadoop is the open-source implementation of the three Google’s technologies. Hadoop contains a series of sub-projects, in which HDFS corresponds to GFS; MapReduce corresponds to Google’s MapReduce; HBase corresponds to BigTable. Hadoop has been supported by many researchers and big companies, although it’s still on a development stage. On Hadoop’s sponsors list, we can find Microsoft, Google, Yahoo!, AMD, HP, IBM, etc., all of which are leaders of the industry. As a open-source project, Hadoop has to thank researchers in universities all over the world as well. Now Hadoop has more than 10 sub-projects.

Developing distributed applications is always time consuming. To reduce our work load, we didn't use Java to write distributed applications, like MapReduce programs, when we did our experiments. Instead, we used a Clojure-based query language for Hadoop: Cascalog. It's similar to Datalog language. For example, in Appendix 1, the Cascalog format of the ninth query is: [?0 ?1 ?2] (Student ?0) (Faculty ?1) (Course ?2) (advisor ?0 ?1) (takesCourse ?0 ?2) (teacherOf ?1 ?2).

Query execution using non-SQL queries has been an important application area of Hadoop. In the following, I will discuss the execution and its efficiency of LUBM queries on Amazon. The director of our project Tomasz Wiktor Wlodarczyk and my classmate Yi Han compared different configurations and try to find out the factors that may affect query efficiency. They used three of the LUBM test queries, which represent low selectivity, high selectivity and complex query respectively. Following part of this section is based on their paper "Performance Analysis of Hadoop for Query Processing" [41].

At first I will explain how to execute a Cascalog query on Amazon. On Amazon, we can run "jar" package, and as mentioned above, Cascalog is based on Clojure, which is run in Java Runtime Environment. Hence, what we need to do is that first edit Clojure file, which contains Cascalog queries, and then compile the file into a "jar" package and run it on Amazon. As a distributed query language for Hadoop, we don't have to consider problems concerning distribution. The Clojure file corresponding to LUBM Query 2 can be found in Appendix 3. After we have the Clojure file, we need a tool, Leiningen, to compile it into a "jar" package. (The usage of Leiningen can be found at https://github.com/technomancy/leiningen#readme.) The following will introduce experiment configurations and result analysis.

(1) Computing Cluster Configuration: the number of nodes used counts from 2 to 20. One thing that needs to be noticed is that since 1 of the nodes is used as a name node, the actual number of computing nodes is one less than the total number. Each node is an Amazon virtual machine. In the experiments, we used both m1.small instances and m1.large instances. The difference between those two is that m1.large instances have better I/O performance, and other parameters such as CPU, memory and storage are also better. However, I/O performance seems the key factor that affects our results. We tested two different configurations for the number of Reducers: 1 Reducer and the number of nodes – 1 Reducers, so that we can clearly see how the number of Reducers affects different queries.

(2) Query and Data Configuration: as mentioned above, we will use LUBM's Ontology, data and queries. LUBM is usually used for triple-based test, so the data generated is mainly RDF triples, and the test queries also need some reasoning. At this stage of our research, our purpose is to test the execution efficiency of queries in distributed environment, so we choose only the 1st, 2nd, and the 14th query. The 1st query accords to high selective queries, which needs to read a lot of data when execution, while 2 data files. The 2nd query accords to complex relations between multiple data files. The 14th query accords to low selectivity. Our tests are based on 6 different data sets: 50, 100, 200, 500, 1000 and 6000 universities. The smallest file in these data sets is a file needed by Query 2, in the 50-universities data set, which has size 13.6 MB; the biggest one is a file in the 6000-universities data set needed by Query 14, which has size 2.9 GB. The queries are executed by Cascalog.

The tests results and analysis are below. Figure 3.1 to3.5 are from "Performance Analysis of Hadoop for Query Processing".

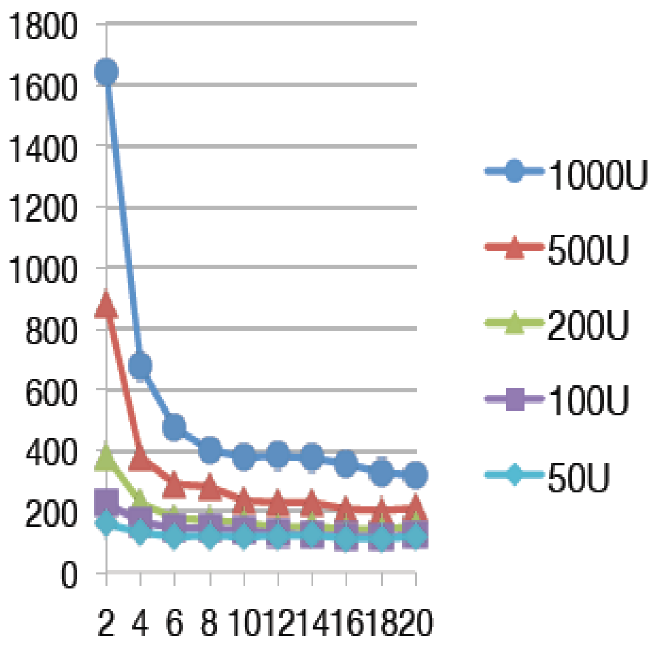


Figure 3.1

Figure 3.1 shows the execution time of Query 1 under each data set when 1 Reducer and m1.small instances are used. Similar results can be found when (number of nodes)-1 Reducers are used. Then for high selective query, the number of Reducers has no obvious effect on query efficiency. In the mean while, their tests also show that when 2 to 4 nodes are utilized, the usage of high I/O performance nodes can effectively reduce querying time.

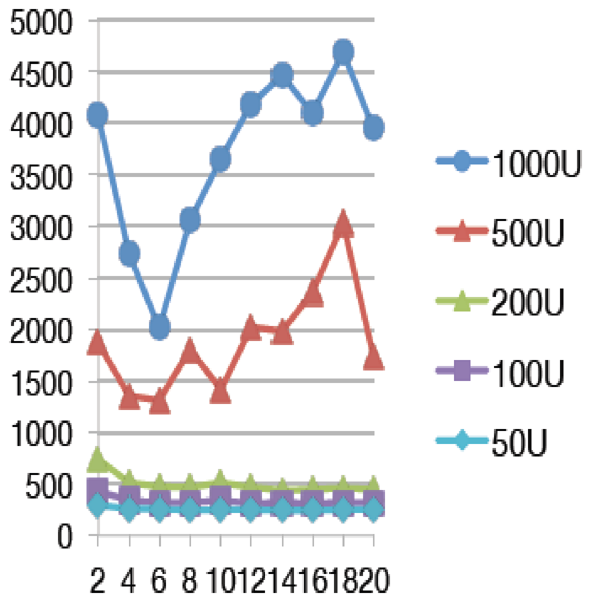


Figure 3.2

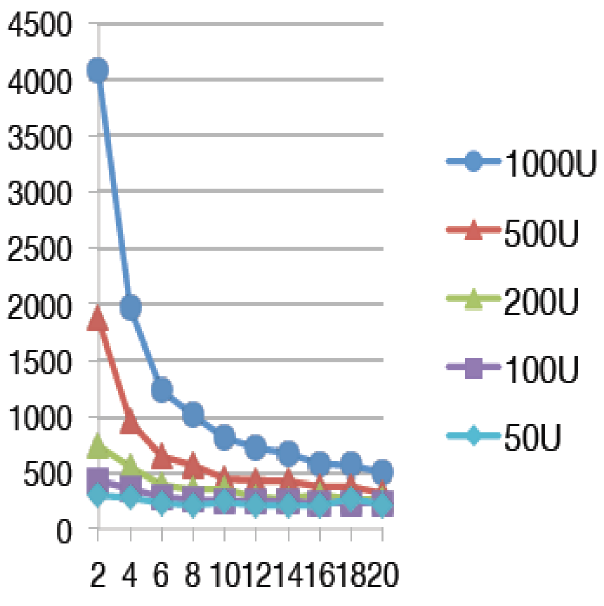


Figure 3.3

Figure 3.2 shows the execution time of Query 2 under each data set when 1 Reducer and m1.small instances are used. Figure 3.3 shows the execution time of Query 2 under each data set when (number of nodes)-1 Reducers and m1.small instances are used. From these figures we can see: low selective query has similar characters as high selective query, and by carefully observing, we see that I/O performance doesn't equal to network and disk performance.

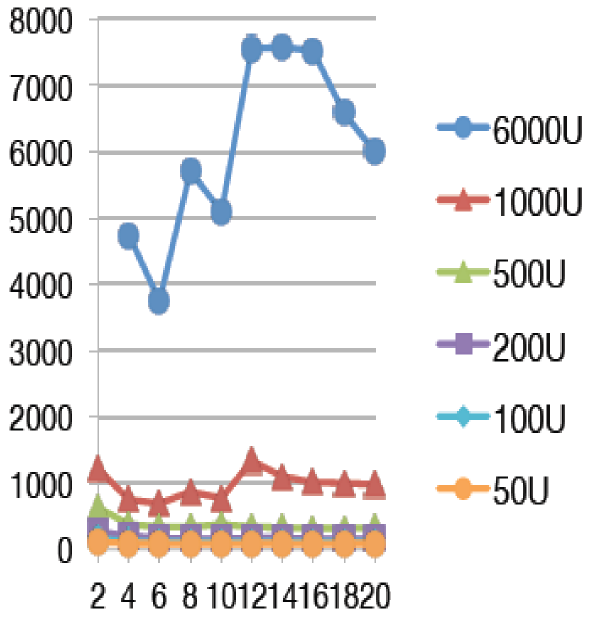


Figure 3.4

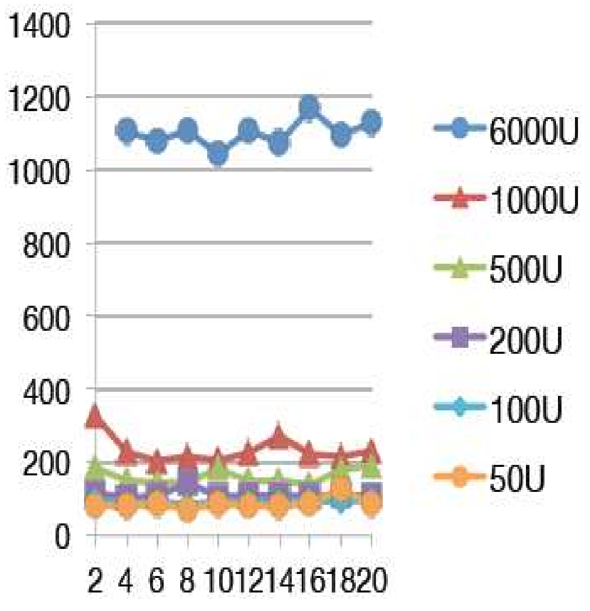


Figure 3.5

Figure 3.4 shows the execution time of Query 14 under each data set when 1 Reducer and m1.small instances are used. Figure 3.5 shows the execution time of Query 14 under each data set when 1 Reducer and m1.large instances are used. Compare them we can see: when only 1 Reduceer is utilized, using m1.large instances can avoid performance decrease caused by usage of more nodes. On the other hand, the increase of number of nodes does not bring improvement of performance. This may be because of the limit of disk capacity, since only one Reducer is used to write relatively large output to HDFS.

Though the tests above, we get to know the factors that affect query efficiency in distributed environment. In our following research, we will use similar comparison method to make further tests.

3.2 Data Preparation

Using data generator provided by LUBM, we can get a large mount of data about the university semantic. For following test use, we generated data for 1, 2, 5, 10, 50, 100, 200, and 500 universities. The data generated is in OWL format. In order to use our query mechanism, we need to transform it into text files [42]. The organization methods of these text files are: the instances of a class will be stored in a text file, whose name is the class name; the instances of a resource will be stored in a text file, whose name is the resource name, and each line in the file is a tuple of the subject and object of the resource. For example, "Course" is the class for courses, and some lines in its text file are listed below:

http://www.Department0.University0.edu/Course0

http://www.Department0.University0.edu/Course1

http://www.Department0.University0.edu/Course2

http://www.Department0.University0.edu/Course3

http://www.Department0.University0.edu/Course4.

Some lines in the text file of resource "advisor" are:

http://www.Department0.University0.edu/UndergraduateStudent4
http://www.Department0.University0.edu/AssistantProfessor4

http://www.Department0.University0.edu/UndergraduateStudent10
http://www.Department0.University0.edu/AssistantProfessor6

http://www.Department0.University0.edu/UndergraduateStudent18
http://www.Department0.University0.edu/FullProfessor7。

Table 3.1 lists the size of each data set. We can see from the table, the size will be very large when the data is for more than 100 universities. It's almost impossible to process data like this if distributed computing and storage is not utilized.

Table 3.1 University Data Sets Sizes

Data Set	**Size**
1 University(uni1)	9.72MB
2 Universities(uni2)	22.7MB
5 Universities (uni5)	61.7MB
10 Universities (uni10)	126MB
50 Universities (uni50)	644MB
100 Universities (uni100)	1.28GB
200 Universities (uni200)	2.62GB
500 Universities (uni500)	13.6GB

3.3 Materialization

We used Java-based framework Jena to materialize data. Except for loading, manipulating, and querying Ontology API, Jena also provides some built-in reasoners: RDFS reasoner supports reasoning to RDF documents; OWL reasoner supports reasoning to OWL documents, and it's more powerful but less efficient because of its computing complexity. The procedure of our materialization is: loading LUBM Ontology definition (http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl), loading data generated by LUBM data generator, running reasoner and getting reasoned Ontology model, and then write data in the new model into text files.

In order to select a high-performance reasoner, we took a test: materializing data of 10 universities using different reasoners, and comparing execution time of each reasoner. Here we added an external reasoner Pellet. Pellet provides a easy-to-use Java API, which we can invoke under Jena framework. The test results are listed in Table 3.2. We can see from the table that the built-in reasoner of Jena is very inefficient, so in our following research, we will choose only Pellet.

Table 3.2 Reasoner Performance Test

Reasoner Type	**Time Consumption Materializing Uni10**
RDFSReasoner	6 Hour 42 Min 5 Sec
OWLMicroReasoner	9 Hour 15 Min 41 Sec
Pellet	10 Min 12 Sec

With the combination of Jena and Pellet, we materialized each data set mentioned in Chapter 3.2. One thing that has to be mentioned is that like other AI and Semantic Web applications, because of the computing complexity, the computing ability and memory (usually the bottleneck) of a normal local computer can only afford to materialize data of 10 universities. For larger size of data, for example 500 universities, we have to splite it into 50 parts. Each part contains data of 10 universities. Here we assume that there is no semantic relation between any two universities.

We can see from the generated text files, that after reasoning, the number of instances (materialized data) in the Ontology model increases. Table 3.3 lists the size difference of each data set before and after reasoning. In the origin data set of 10 universities, "course.txt" contains only directy instance of class "Course", and the file size is 495KB. In the materialized data set, the size of "course.txt" is 1066KB. It contains not only direct instances of "Course", such as http://www.Department0.University0.edu/Course3, but also instances of class "GraduateCourse", such as http://www.Department14.University5.edu/GraduateCourse9. In Figure 3.1, we see "GraduateCourse" is a subclass of "Course", so the instances of "GraduateCourse" should also be the instances of "Course". Finding out implicit data in an Ontology is the main objective of materialization.

Table 3.3 Data Set Size Differences

Data Set	**Size Increment**	**Triple NO. Increment**
uni1	1274.46KB	21874
uni2	2889.43 KB	48894
uni5	7806.16 KB	131183
uni10	15943.12 KB	267207
uni50	81447.77 KB	1347650
uni100	163040.36 KB	2759975
uni200	339364.64 KB	5556674
uni500	858052.28 KB	13974003

In Computer Science, we often have to make trade-off between time and space. That is: smaller storage space always results in longer computing time, and in order to achieve higher computing speed, we need to sacrifice some extra space. Materailization reasons the orginal Ontology, and adds inferred data into orginal data set, which sacrifices space, but saves reasoning time when semantic querying.

3.4 Query Rewriting

We use Requiem developed by Computing Lab of Oxford University to implement query rewriting. Requiem loads an Ontology definition, and then rewrites an input datalog query. Down below I will introduce its query rewriting algorithm.

The input of the algorithm is a conjunctive query Q and a TBox T (Ontology definition), and the output is a series of queries rewritten from Q and T. The algorithm first turns Q and T into a series of clauses, and then gets new clauses based on resolution computing, as shown below: [43]

```
Input: conjunctive query Q, DL-Lite TBox
R = $\Xi(T) \cup \{Q\}$;
repeat
(saturation) forall clauses C1 , C2 in R do
R = R ∪ resolve(C1 , C2 );
end
until no new clause resoluted
return $\{C \mid C \in$ unfold(ff(R)), and C has the same head predicate as Q$\}$;
```

The algorithm consists of 4 steps: clausification, saturation, unfolding and pruning. Clausification transforms Q and T into a series of sets of clauses $\Xi(T) \cup \{Q\}$. $\Xi(T)$ is the clause set from T. The mappings from axioms in TBox to logic clauses are shown in Table 3.4.

Table 3.4 Mappings from T to $\Xi(T)$

DL-Lite Clause (**$\Xi(T)$**)	**DL-Lite Axioms** (**T**)
$B(x) \leftarrow A(x)$	$A \sqsubseteq B$
$P(x, f(x)) \leftarrow A(x)$	$A \sqsubseteq \exists P.B$
$B(f(x)) \leftarrow A(x)$	

DL-Lite Clause (Ξ(T))	DL-Lite Axioms (T)
$P(f(x), x) \leftarrow A(x)$	$A \sqsubseteq \exists P^- .B$
$B(f(x)) \leftarrow A(x)$	
$A(x) \leftarrow P(x, y)$	$\exists P \sqsubseteq A$
$A(x) \leftarrow P(y, x)$	$\exists P^- \sqsubseteq A$
$S(x, y) \leftarrow P(x, y)$	$P \sqsubseteq S, P^- \sqsubseteq S^-$
$S(x, y) \leftarrow P(y, x)$	$P \sqsubseteq S^-, P^- \sqsubseteq S$

In the step of saturation, the algorithm keep resolving any two elements in the clause set repeatedly to get new clauses. The function “resolve” takes two clauses C_1 and C_2 as input, resolves new clauses by looking for feasible matches in the reasoning template shown in Figure 3.6, and returns the set of all new clauses. If no matches found, it returns empty set.

$$\frac{C(x) \leftarrow B(x) \quad B(f(x)) \leftarrow A(x)}{C(f(x)) \leftarrow A(x)}$$

$$\frac{B(x) \leftarrow P(x,y) \quad P(x,f(x)) \leftarrow A(x)}{B(x) \leftarrow A(x)} \qquad \frac{B(x) \leftarrow P(x,y) \quad P(f(x),x) \leftarrow A(x)}{B(f(x)) \leftarrow A(x)}$$

$$\frac{B(x) \leftarrow P(y,x) \quad P(x,f(x)) \leftarrow A(x)}{B(f(x)) \leftarrow A(x)} \qquad \frac{B(x) \leftarrow P(y,x) \quad P(f(x),x) \leftarrow A(x)}{B(x) \leftarrow A(x)}$$

$$\frac{S(x,y) \leftarrow P(x,y) \quad P(x,f(x)) \leftarrow A(x)}{S(x,f(x)) \leftarrow A(x)} \qquad \frac{S(x,y) \leftarrow P(x,y) \quad P(f(x),x) \leftarrow A(x)}{S(f(x),x) \leftarrow A(x)}$$

$$\frac{S(x,y) \leftarrow P(y,x) \quad P(x,f(x)) \leftarrow A(x)}{S(f(x),x) \leftarrow A(x)} \qquad \frac{S(x,y) \leftarrow P(y,x) \quad P(f(x),x) \leftarrow A(x)}{S(x,f(x)) \leftarrow A(x)}$$

$$\frac{Q_P(\vec{u}) \leftarrow B(t) \wedge \bigwedge D_i(\vec{t_i}) \quad B(f(x)) \leftarrow A(x)}{Q_P(\vec{u})\sigma \leftarrow A(x)\sigma \wedge \bigwedge D_i(\vec{t_i})\sigma}$$
where $\sigma = \mathsf{MGU}(B(t), B(f(x))$, and $B(t)$ is deepest in its clause.

$$\frac{Q_P(\vec{u}) \leftarrow P(s,t) \wedge \bigwedge D_i(\vec{t_i}) \quad P(x,f(x)) \leftarrow A(x)}{Q_P(\vec{u})\sigma \leftarrow A(x)\sigma \wedge \bigwedge D_i(\vec{t_i})\sigma}$$
where $\sigma = \mathsf{MGU}(P(s,t), P(x,f(x))$, and $P(s,t)$ is deepest in its clause.

$$\frac{Q_P(\vec{u}) \leftarrow P(s,t) \wedge \bigwedge D_i(\vec{t_i}) \quad P(f(x),x) \leftarrow A(x)}{Q_P(\vec{u})\sigma \leftarrow A(x)\sigma \wedge \bigwedge D_i(\vec{t_i})\sigma}$$
where $\sigma = \mathsf{MGU}(P(s,t), P(f(x),x)$, and $P(s,t)$ is deepest in its clause.

Figure 3.6 Reasoning Template of resolve function

The next step is unfolding. Function “ff” eliminates functions of all clauses, and function “unfold” unfolds clauses in a set. For example, a clause set N = {$Q_P(x) \leftarrow A(x)$, $A(x) \leftarrow B(x)$}, and then unfold(N) = N ∪ $\{Q_P(x) \leftarrow B(x)\}$.[44] Finally, in step pruning, all clauses which have different head predicates from Q are abandoned.

In the following, an example will explain the application of query rewriting. [45]

Assume there is a table “Professor” in a relational database, which contains attributes: “Name”,”Faculty”, “Telephone”; another table “Student”, which contains attributes: “Name”, “Professon”, “Address”, “Tutor”. We can find a suitable Ontology to define the data semantic. For instance, we use following OWL 2 QL Ontology Q to describe the database DB:

$$\mathrm{Pr}\,\mathit{ofessor} \subseteq \exists \mathit{teaches} \qquad (1)$$

$$\exists \mathit{teaches} \subseteq \mathit{Teacher} \qquad (2)$$

$$\exists \mathrm{hasTutor}- \subseteq \mathrm{Professor} \qquad (3)$$

Axiom (1) indicates that a “Professor” should at least “teach” someone; axiom (2) indicates that the subject of resource “teach” is “Teacher”; axiom (3) indicates that the domain of resource “hasTutor” is “Professor”.

Once we can map the classes and resource in the Ontology into DB, the query to Ontology data can be transformed into query to DB. The advantage of doing this is, on one hand, users can query according to the semantic of data, on the other hand, DB provides efficient management to data. The mapping from Ontology to database can be expressed as D->Q_D, in which D is classes and resources in the Ontology, and Q_D is a query to the database. In our example, the mapping M is defined as:

Professor->SELECT Name FROM Professor

hasTutor->SELECT Name, Tutor FROM Student

The query to Ontology data can be divided into two steps: first, rewrite the query into a set of对本 new queries using Ontology definition, and then translate the queries into SQL queries using mapping. For example, consider query Q=Q(x)<-Teacher(x). From Axiom (1) and (2) we know, a professor must be a teacher, so the rewritten queries Q_0 must contain this information, i.e., Q_0 should retrieve all instances of class “Professor” and class “Teacher”. After running through Requiem, Q_0 is:

Q(x)<-Teacher(x) (4)

Q(x)<-teaches(x,y) (5)

Q(x)<-Professor(x) (6)

Q(x)<-hasTutor(y,x) (7)

When we get Q_0, we can start to query on DB. First, we should translate Q_0 into SQL. The translation is simply replace classes and resources in queries according to M. Noticing that there is no mappings about “Teacher” and “teaches” in M, (4) and (5) can be ignored. In the end, the rewritten SQL queries are:

sql(Q_0)=SELECT Name FROM Professor UNION

SELECT Tutor FROM Student.

Using these two SQL, we can get the right semantic data in relational database.

As Requiem only provides reasoning function to TBox, while has no relations to Ontology instance data, so it cannot process queries containing instances. For example, Requiem can process the query Q(?0) <- UndergraduateStudent(?0), while cannot process query Q(?0) <- Person(?0), hasAlumnus(<http://www.University0.edu>,?0). In the 14 test queries provided by LUBM, we rewrite the 2nd, 6th, 9th, and 14th queries, which contain no instance, using Requiem. After rewriting, the 2nd query generates 4 new queries (listed in Appendix 2), the 6th query generates 169 new queries, and the 9th and the 14th query generate only 1 new query separately. As Requiem takes only datalog queries as input, we need to translate queries before and after rewriting.

3.5 Tests

In this chapter we will test the two approaches materialization and query rewriting, and compare their efficiency during semantic data querying. For materialization, because existing reasoner can’t run in distributed environment, that is to say, in an distributed environment, we can’t do runtime reasoning to semantic data (actually in application, people don’t do runtime reasoning for each querying), we assume the semantic data has been materialized, and we use materialized data when test. According to our experience to materialization efficiency (high computing complexity, time consuming), if the approach query rewriting is not too inefficient compared to querying materialized data using original queries, we can conclude that query rewriting is better than materialization in our case. For query rewriting, we use Requiem. In this stage when we haven’t integrate it into our system, we will use rewritten queries directly, and ignore the effect query rewriting process makes to our test results. In fact, according to our experience, for the 4 queries we are going to test, the rewriting is quite efficient, so the ignorance won’t have much effect.

3.5.1 Tests on Local Machine

Because of the computation ability limit, we can only test data for 10 universites the most on a local machine. The test results are shown in Table 3.5.

Table 3.5 Test Results on Local Machine

Query	**Time Consumption of Materialization**	**Time Consumption of Query Rewriting**
Q2	11 h 3 m 7 s	2 m 25 s
Q6	≤1s	61 h 1 m 29s
Q9	10 h 13 m 23 s	2s
Q14	≤1 s	≤1 s

We can see in the table, for Q2 and Q9, query rewriting is much more efficient than materialization. For Q6, query rewriting is much less efficient than materialization. We can find out the reason behind it by studying data and the queries. Take Q9 as an example:

```
SELECT ?0 ?1 ?2
WHERE { ?0 rdf:type :Student .
          ?1 rdf:type :Faculty .
          ?2 rdf:type :Course .
          ?0 :advisor ?1 .
          ?0 :takesCourse ?2 .
          ?1 :teacherOf ?2 }
```

There are 6 restrictions, so the computing complexity is relatively high. For the new query rewritten from it:

```
SELECT ?0 ?1 ?2
WHERE { ?0 :advisor ?1 .
      ?0 :takesCourse ?2 .
        ?1 :teacherOf ?2 }
```

There are only 3 restrictions, which means the computing complexity has been lowered. As for Q6, after written, there are 169 new queries generated, apparently the query efficiency won't be high.

We get an initial conclusion from this test: query rewriting always specifies and refines an input query through Ontology definition, and eliminates redundant or unnecessary restrictions (such as Q9), so each new query will be simpler than the original one. If the new queries are not too many (such as Q6), the efficiency of query rewriting won't be much worse than when querying materialized data, sometimes maybe even better.

3.5.2 Tests in Distributed Environment

After configuring Hadoop framework of virtual machines in AWS, creating Clojure job for each query, and uploading original and materialized data, we can start to test. We use 10 virtual machines and test data for 5, 10, 50, 100, 200, and 500 universities. The tests results are listed in the table below:

Table 3.6 Results for 5 Universites

Query	**Time Consumption for Materialization (s)**	**Time Consumption for Query Rewriting (s)**
Q2	378.578	1760.747
Q6	101.741	45279.46
Q9	509.325	267.384
Q14	83.266	183.029

Table 3.7 Results for 10 Universites

Query	**Time Consumption for Materialization (s)**	**Time Consumption for Query Rewriting (s)**
Q2	408.276	1853.336
Q6	89.139	42805.239
Q9	550.926	274.711
Q14	92.712	103.036

Table 3.8 Results for 50 Universites

Query	**Time Consumption for Materialization (s)**	**Time Consumption for Query Rewriting (s)**
Q2	376.894	1863.806
Q6	97.19	46012.422
Q9	579.287	310.509
Q14	143.797	93.085

Table 3.9 Results for 100 Universites

Query	Time Consumption for Materialization (s)	Time Consumption for Query Rewriting (s)
Q2	422.783	1765.884
Q6	142.15	47550.207
Q9	575.015	335.73
Q14	102.446	101.024

Table 3.10 Results for 100 Universites

Query	Time Consumption for Materialization (s)	Time Consumption for Query Rewriting (s)
Q2	414.057	1946.662
Q6	149.39	54324.935
Q9	608.642	406.283
Q14	112.014	166.759

Table 3.11 Results for 500 Universites

Query	Time Consumption for Materialization (s)	Time Consumption for Query Rewriting (s)
Q2	405.784	2016.911
Q6	152.382	58280.197
Q9	820.153	542.146
Q14	122.382	111.719

From the tables above we can see: because of the differences in architecture and querying mechanism, materialization and query rewriting have different performance from local situation. We compare test results for 10 universities in Table 3.7 with the results in Table 3.5, as shown in Figure 3.7.

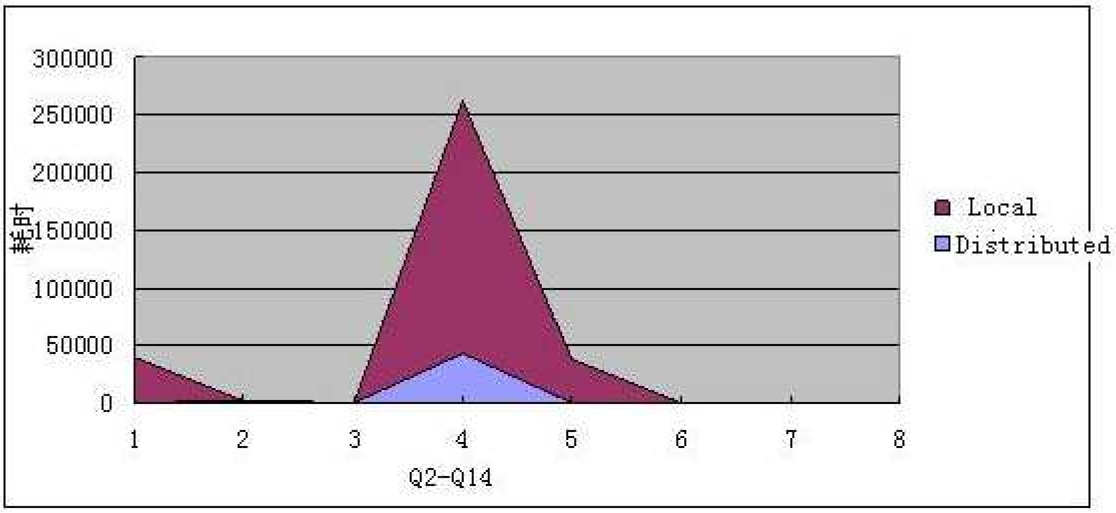


Figure 3.7 Querying Efficiency Comparison between Local and Distributed Situation

Although some operations which are not so time-consuming on local machine (such as querying of rewritten Q2) will spend more time in distributed environment (distributed computing always brings extra cost, such as communication between different nodes and job assignment), from Figure 3.7 we can see, operation which is extremely time-consuming on local machine (such as querying of rewritten Q6) has much better executing efficiency distributedly.

Figure 3.8 is depicted from Table 3.6 to Table 3.11. It doesn't conclude test results for Q6 after rewritten, in order to clearly illustrate other data.

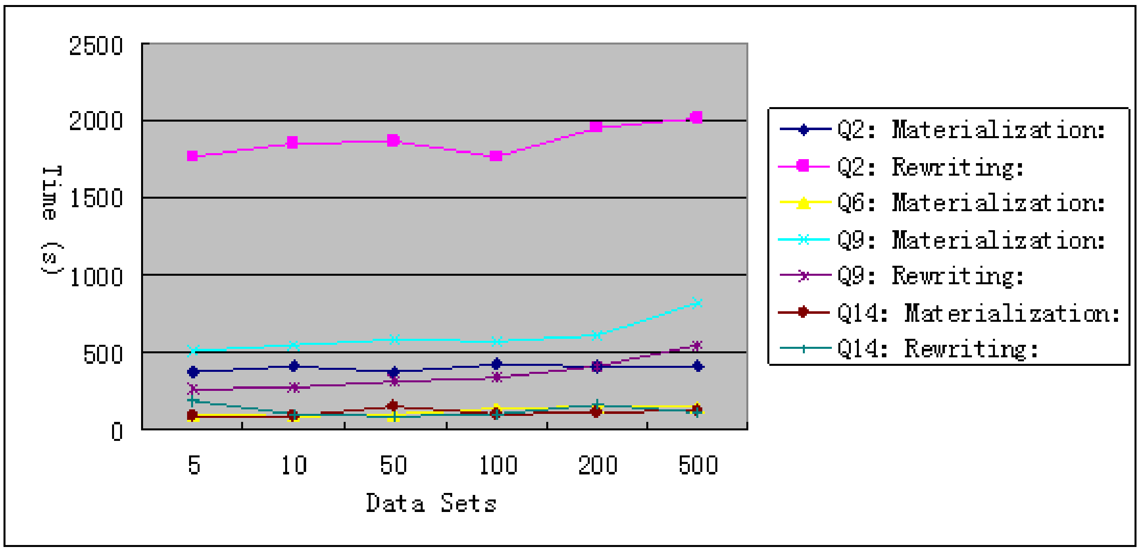


Figure 3.8 Query Efficiency Comparison in Distributed Envrionment

From the figure above we see, except the situation of Q2 rewriting, each query has almost the same query efficiency using either materialization or query rewriting. Query rewriting for both Q2 and Q6 is much less efficient in distributed environment, because there are a lot of new queries generated after rewritten. More over, in distributed environment, with the growth of size of data sets, there's only slight decrease on query efficiency, which is to say, semantic data querying in distributed environment is more efficient to large data sets. If we define query efficiency as average time consumption on each university, Figure 3.9 lets us understand our conclusion better.

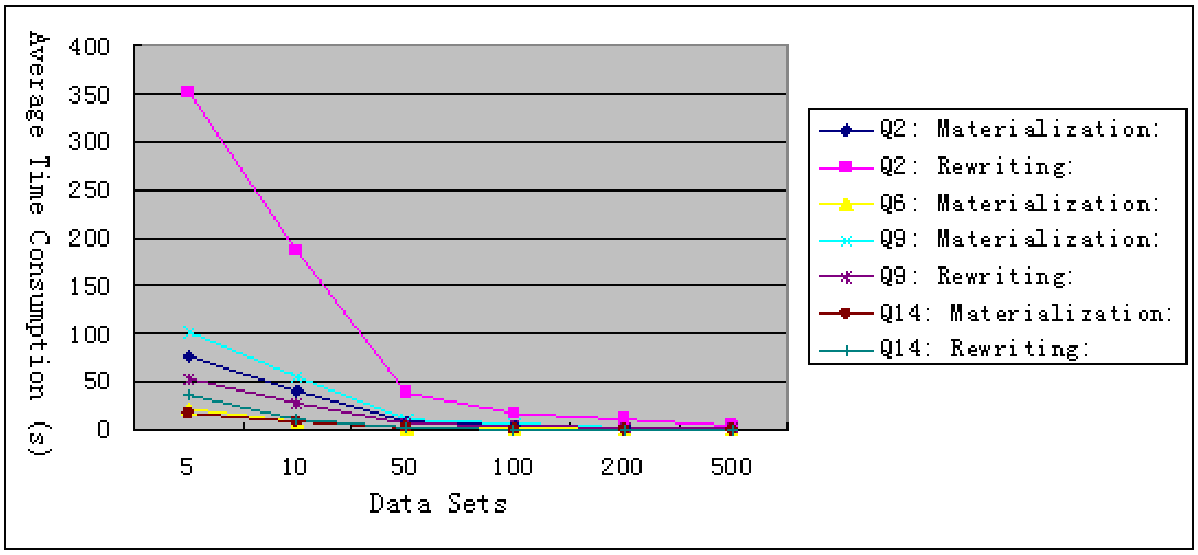


Figure 3.9 Query Efficiency Comparison on Average

3.6 Conclusion

We get following conclusion based on our tests above:

(1) The execution efficiency of querying semantic data with rewritten queries is almost the same as of querying materialized data with original query.

(2) Some of the test queries will have a lot of new queries after rewritten, so there is a relatively big decrease in terms of querying efficiency.

(3) Because of the refinement and simplification of query rewriting, sometimes query rewriting is even more efficient than materialization.

(4) In distributed environment, the querying efficiency will not decrease significantly with the growing of data size, so the larger the data set is, the more advantage distributed querying takes.

Based on the conclusions above, we decide to develop a semantic data querying application based on Amazon cloud computing platform. The purposes of the development are: to gain experience of developing distributed semantic data querying applications, and to support other Semantic Web researches using this application.

Chapter 4 Application Development

We developed a semantic data querying application based on Amazon cloud computing service for the main data source we use for Semantic Web research: LUBM. There are two premises:

(1) The distributed environment infrastructure consists of Amazon S3 and EC2, and underlying software framework is Hadoop.

(2) Semantic data is stored in text files; instances of a class are stored in a text file whose name is the class name, and each line of which represents an instance; instances of a property are stored in a text file whose name is the property name, and each line of which represents the relation between two class instances.

4.1 Design

The application has two main parts: data querying and data management. In data querying, a user submits a SPARQL or SQWRL query, and the application processes it and tranforms it into Cascalog. Then it uploads the query to cloud end and executes the query. Query results and logs will be returned or stored. In data management, a user can use the application to add, delete files, and add semantic data. The application logic is shown in Figure 4.1.

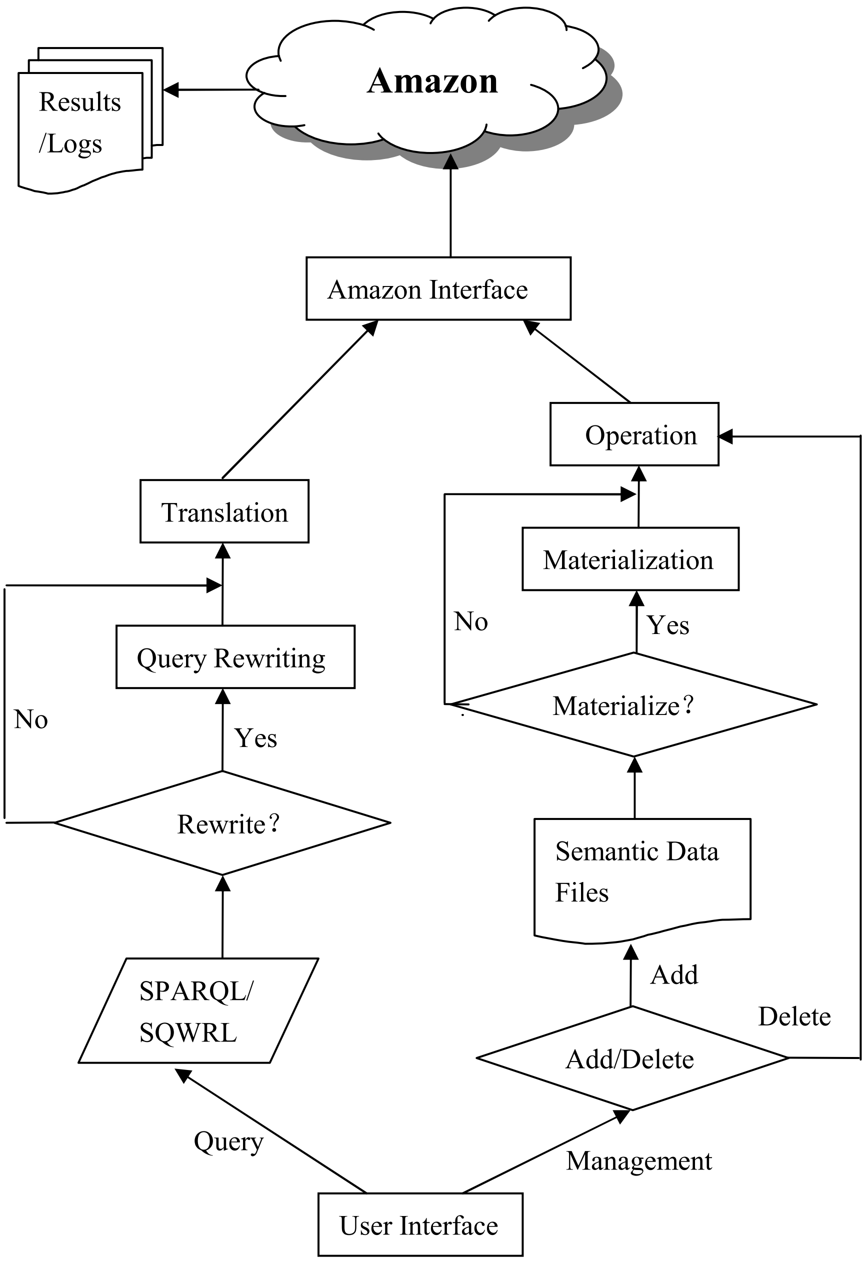


Figure 4.1 Application Logic

In Figure 4.1, the left part is data querying, and the right part is data management. When querying, the user selects one of the SPARQL or SQWRL queries stored in a file. If the user needs to rewrite the query, then the application goes into its query rewriting module; or it goes into translation module. The translation module get original or rewritten query, translates it into Clojure-based Cascalog, and then uploads it onto Amazon to execute the query. As mentioned before, the reason why we translate queries into Cascalog is because Cascalog is for Hadoop, and its query efficiency to test files is very high. For data management, deleting a file is simple. The application can directly delete a file object on Amazon S3. While adding data, the application first decides if the file which data should be added to exists. If not, then the application creates a new file object; and if yes, the application needs to add data into the file. Here one thing that we need to consider is that Amazon S3 supports only creating and deleting file objects, but does not support modifying a file. Before adding data, the user can choose whether to reason and materialize data. According to user's need, the materialization module will provide two reasoners: RDFS reasoner and OWL DL reasoner. Querying and Management Module both need an interface to interact with Amazon cloud environment remotedly.

After all the experiments and tests we did, we have actually implemented most of the key functions, such as query rewriting and materialization. The remaining work is to integrate our previous work and execute distributed Amazon jobs locally.

4.2 Implemetation

The application is implemented with Java. We didn't develop GUI for now, and it will be published as executable package: dsts.jar. The input parameters of the application are:

-q file-with-queries.txt(query mode; "file-with-queries.txt" is the text file which stores the queries that may be used. Each line is a query, which can be commented by "//".)

- -o=x (optional; indicate the directory "x" where query results are stored in query mode)
- -l=x (optional; retrieve only "x" records of all the results in query mode)
- -r (optional; used when need to rewrite the input query)

-e (management mode)

- -d=x (optional; delete file "x" in management mode)
- -a=x (optional; add file "x" into data set in management mode)
- -ao=x (optional; add semantic data file (.owl) "x" into data set in management mode)
- -rdfs (optional; materialize added data using RDFS reasoner in management mode)
- -dl (optional; materialize added data using OWL DL reasoner in management mode)

-s=x (store statistics and logs into file "x")

In the following I will explain the implementation of the application by modules.

(1) Query transformation: In our design (Figure 4.1), users can input 2 types of queries: SPARQL and SQWRL. The input and output queries of Requiem are all in Datalog format. The input to translation module is in SQWRL format. Hence we created a "QueryTranslator" class to

translate SPARQL to Datalog, SQWRL to Datalog, Datalog to SQWRL, and SPARQL to SQWRL. Some examples for these formats are below:

SPARQL：

```
SELECT ?1 ?0
WHERE { ?0 rdf:type :Publication . ?1 rdf:type :AssistantProfessor . ?0: :publicationAuthor ?1 }
```

SQWRL：

```
Publication(?0) ^ AssistantProfessor(?1) ^ publicationAuthor(?0,?1) -> sqwrl:select(?1,?0)
```

Datalog：

```
Q(?1,?0) <- Publication(?0), AssistantProfessor(?1), publicationAuthor(?0,?1)
```

(2) Query Rewriting：implemented in class “QueryRewriter”. This module utilizes functions given by Requiem. Function getQuery() transforms an input Datalog query into a “Clause” class to be used by rewriter. Function rewriter() is the main method to rewrite a query. It has two parameters: one is the query in Datalog format, and the other is mode identifier. Requiem has 3 modes: N (Naïve), F (Full), and G (Greedy). The three modes have different rewriting efficiency and make different optimization (different number of queries after rewriting). We use G mode only. The function reads in the query to be rewritten, runs the algorithm, and then returns a String type List object, which contains new Datalog queries generated.

(3) Translation: The whole application accepts SPARQL and SQWRL queries, and they will be transformed into Datalog format after rewritten, but they won’t be used directly to execute. Before deployment, they will be translated into Cascalog. This is done by a convertor. After that, the Cascalog object will be transformed into executable Clojure code depending on the deployment environment.

(4) Amazon Interface: The Amazon interface is implemented mainly using APIs provided by Amazon S3, EC2, SimpleDB, etc.

In our application, we need to upload, download and delete files on Amazon S3. No matter uploading, downloading or deleting, we first need to generate a Amazon S3 object using Amazon account information. Then using fuctions putObject(), getObject() and deleteObject() of the object, we can upload, download and delete files.

Except for file operations, in order to execute remote queries, we need to create virtual machine, configure runtime environment, and execute queries using Amazon API. These can be divided into 5 steps: deployment plan, installation, execution, result retrieving, and clearing. In deployment plan step, the application will generate configuration information needed when executing queries, such as data file to be uploaded, executable package directory and execution parameter, etc. Because difference in runtime environment (AWS or pure Hadoop clusters), the same query will generate different deployment settings. Installation means installing deployment settings and files generated in deployment plan step into runtime environment. If possible, installation may change some of the parameters to adapt current installation. Execution means executing queries according to deployment plan and installation parameters. Result retrieving copies query results to local machine. Clearing means clearing changes to environment caused by deployment, installation and execution.

(5)Materialization: Class "Materialization" implements data materialization module. The main fuction in the class is materialization(), whose parameters are: the directory of data to be materialized, reasoner type (0 for RDFS reasoner, 1 for OWL DL reasoner), directory where materialized data to be stored. The function makes use of Jena framework. RDFS reasoner is the builtin RDFS_FULL reasoner, and OWL DL reasoner is Pellet reasoner, which we have proven to be efficient. The function logic is: first reads Ontology definition through URL, and create Jena Ontology model; reads all instance data and adds into the Jena model; runs the reasoner, gets reasoned model; and at last, traverses the whole model, outputs all instances. Jena provides an API to traverse all instances of classes, but no API to traverse property instances. Fuction query() implements traversing all instances of a property. It takes a property name as input, executes following query to Jena model:

```
SELECT ?subject ?object WHERE { ?subject :property ?object }
```

And then it returns the result set of quering. We can get all instances of the property through the result set. As mentioned before, the text file which stored instances has a class or property name as its file name. Function resourcename_filename_Conv() implements the conversion between class or property name and text file name.

(6) Adding and deleting data: Data to be added can be in two formats: one is text files as we have already had; the other is OWL files which contain semantic data. In case when we need to materialize added data, as we mentioned above, in either format, the application first reads all data into Jena Ontology model, and then writes reasoned data out to text files. In case when no need to materialize, for data in text files, we can add directly; for data in OWL format, it will be read into Jena Ontology model, and then written out to text files to be added. There are 3 steps when the application add text files to data set stored on Amazon S3: first downloading relevant files on

Amazon S3 (relevant files are those which have the same file name as the ones to be added), then combining files with the same file name, and at last uploading the combined files. The reason why we need the 3 steps is that Amazon S3 doesn't support file updating except creating and deleting. Thus when updating a file, we have to download the existing file, update it locally and then upload it onto S3 and replace the old one. Deleting a file is as simple as sending a deleting requet with the name of the file.

The data adding method described above is obviously inefficient. When the data set grows to a large size, even one file in it may be very large (hundreds of MBs). Downloading and uploading files like this does not only take time, but also takes user's bandwidth. To solve this problem, we consider two possible approaches. One is to create an independent application to update files. The application will be stored on Amazon S3. When a user adds data, a virtual machine is created, and the application is runn on it to finish updating work. The updating work is essentially the same as the process we described above: downloading file, updating, uploading and replacing. The advantage is it doesn't take user's bandwidth, and the interaction between S3 and Amazon virtual machine will be much higher. The disadvantage is that everytime when a user adds data, a virtual machine will be created. When there are a lot of users, it will be costly. The other approach is to divide big files into segments. Limit the file size of each segment (for example 50M). Segments of a file have the same file name prefix, and the suffixes indicate the sequence number of segments. The advantage of this approach is it doesn't consume much extra distributed resource. The disadvantage is it needs more complex file management, so the query efficiency may drop. Because our purpose is to study the common way of implementing similar applications, we didn't solve this problem caused by Amazon S3 character in this research.

4.3 Test

Down below we will test the application.

First test query mode. Unlike the comprehensive tests in Chapter 4, because we just need to test if the application works correctly, here we use 2 m1.small Amazon instances, data for 10 universities.

(1) Without query rewriting: Avoiding "-r" parameter of executable package "dsts.jar", we test the 2nd, 6th, 9th, 14th queries. The results are shown in Table 4.1. The results show that each query can be inputed locally and executed remotely through our application.

Table 4.1 Query Mode Test (Without query rewriting)

Query	Time Cost (s)	Result
Q2	550.715	Correctly finished.
Q6	64.926	Correctly finished.
Q9	528.742	Correctly finished.
Q14	163.088	Correctly finished.

(2) With query rewriting: Adding "-r" parameter of executable package "dsts.jar", we test the 2nd, 6th, 9th, 14th queries. The results are shown in Table 4.2. The results show that the 2nd, 9th, and 14th queries can be executed correctly through our application, but the 6th can't. Checking the exception information, we found it's because the size of parameters overflew when AWS API sending request. We knew that for Query 6, there were 169 new queries generated after rewritten. Through our further test, AWS API can support about 100 queries at most at a time.

Table 4.2 Query Mode Test (With query rewriting)

Query	Time Cost (s)	Result
Q2	1755.915	Correctly finished.
Q6		Exception (AWS Error Message: Size of an individual step exceeded the maximum allowed)
Q9	455.186	Correctly finished.
Q14	129.402	Correctly finished.

Management mode will be tested next.

(1) Adding text file (-a): Select a text file "assistant_professor.txt" (8.6KB) from a local data set and add.

Without materialization (avoiding "-rdfs" or "-dl" in application parameters), run "dsts.jar". The size of file "assistant_professor.txt" on Amazon S3 becomes 8.6KB bigger than before, so we can say it succeeds.

Now we will test it with data materialization. First use RDFS reasoner. After running "dsts.jar" with parameter "-rdfs", compare the new data set with before, and we found that there are 4 files that are 8.6KB bigger: "assistant_professor.txt", "employee.txt", "faculty.txt" and "professor.txt".

This shows us the meaning of materialization. Then we test OWL DL reasoner. After running "dsts.jar" with parameter "-dl", compare the new data set with before, and we found that there are 5 files that are 8.6KB bigger: "assistant_professor.txt", "employee.txt", "faculty.txt", "person.txt", "professor.txt". Compared to RDFS reasoner, there is one more file with size growth. It shows that OWL DL reasoner has stronger reasoning ability than RDFS reasoner.

The tests above show our application works correctly when adding text file with or without materialization.

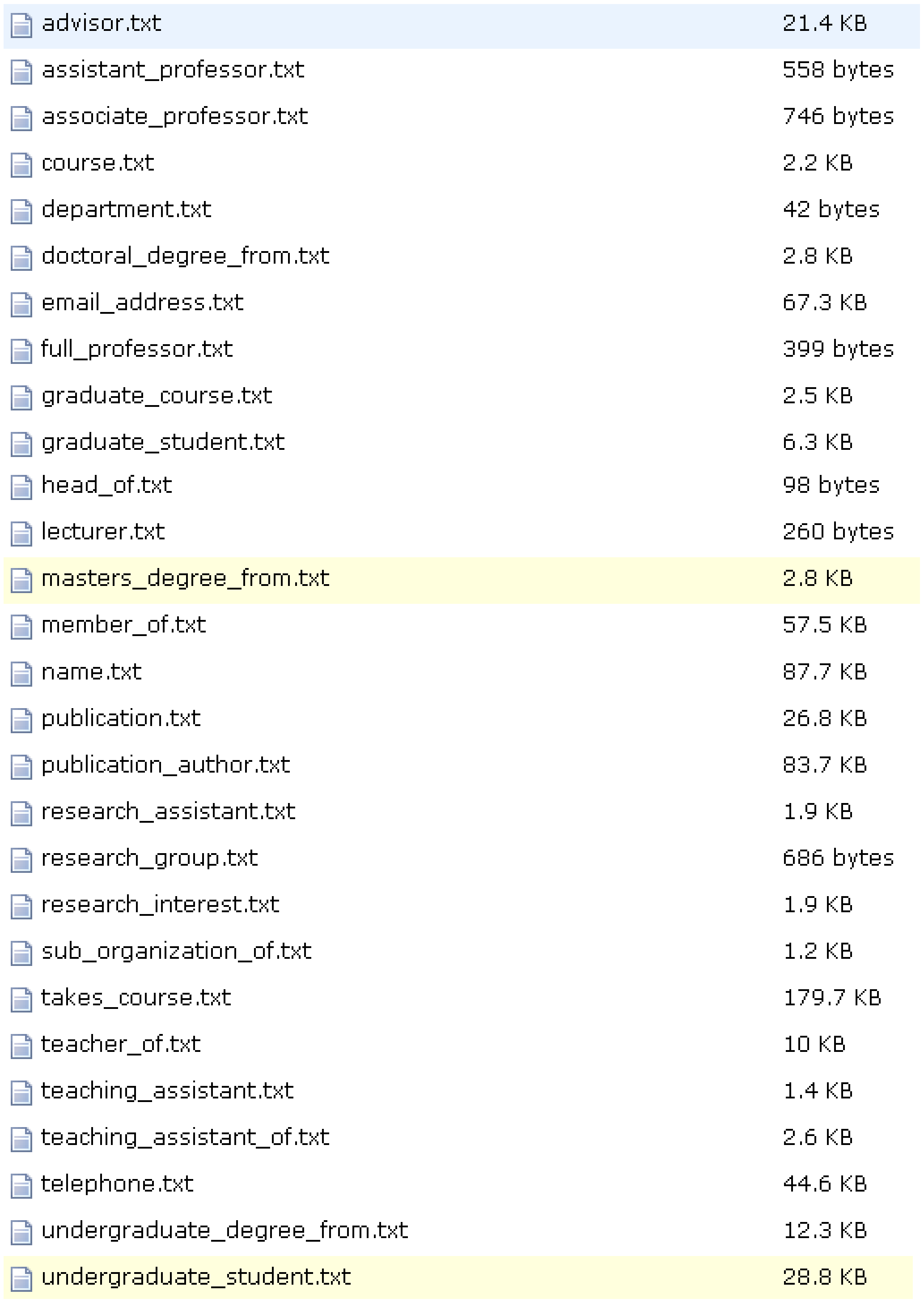
(2) Adding OWL file (-ao): Normally adding a text file only affect one specific class or property, or several classes or properties after materializastion. An OWL file usually contains data about multiple classes and properties, so after loaded by our application, there may be a lot of text files. Select an OWL file (University500_0.owl, 563KB) generated by LUBM data generator as data source to be added.

Without materialization (avoiding "-rdfs" or "-dl" in application parameters), run "dsts.jar". The data set generated from the OWL file is shown in Figure 4.2. The total size of the files is 648.1KB.

Now we will test with data materialization. First use RDFS reasoner. Run "dsts.jar" with parameter "-rdfs". Comparing the files in the data set generated with the one without materialization, many files are larger in terms of size. "course.txt" is 4.8KB larger; "employee.txt" is 1.9KB larger; "faculty.txt" is 1.9KB larger; "organization.txt" is 6.2KB larger; "person.txt" is 37.1KB larger; "professor.txt" is 1.6KB larger; "student.txt" is 30.8KB larger; "university.txt" is 5.5KB larger; "work.txt" is 4.8KB larger; "works_for.txt" is 3.2KB larger. In total, the data set is 95.6KB larger than the one without materialization. Then we test OWL DL reasoner. Run "dsts.jar" with parameter "-dl". Comparing the files in the data set generated with the one without materialization, "chair.txt" is 57B larger; "course.txt" is 4.8KB larger; "employee.txt" is 3.9KB larger; "faculty.txt" is 1.9KB larger; "organization.txt" is 6.2KB larger; "person.txt" is 37.1KB larger; "professor.txt" is 1.6KB larger; "student.txt" is 35.2KB larger; "university.txt" is 5.5KB larger; "work.txt" is 4.8KB larger; "works_for.txt" is 3.2KB larger. In total, the data set is 102KB larger than the one without materialization. Compared with the one using RDFS reasoner, the data set is 6.4KB larger.

The tests above show that our application works correctly when adding OWL file with and without materialization.

(3) Deleting file (-d): Deleting a file is simple. Run “dsts.jar” with parameter “-d” followed by the directory of file to be deleted on Amazon S3. Our test shows the function works correctly.



advisor.txt	21.4 KB
assistant_professor.txt	558 bytes
associate_professor.txt	746 bytes
course.txt	2.2 KB
department.txt	42 bytes
doctoral_degree_from.txt	2.8 KB
email_address.txt	67.3 KB
full_professor.txt	399 bytes
graduate_course.txt	2.5 KB
graduate_student.txt	6.3 KB
head_of.txt	98 bytes
lecturer.txt	260 bytes
masters_degree_from.txt	2.8 KB
member_of.txt	57.5 KB
name.txt	87.7 KB
publication.txt	26.8 KB
publication_author.txt	83.7 KB
research_assistant.txt	1.9 KB
research_group.txt	686 bytes
research_interest.txt	1.9 KB
sub_organization_of.txt	1.2 KB
takes_course.txt	179.7 KB
teacher_of.txt	10 KB
teaching_assistant.txt	1.4 KB
teaching_assistant_of.txt	2.6 KB
telephone.txt	44.6 KB
undergraduate_degree_from.txt	12.3 KB
undergraduate_student.txt	28.8 KB

Figure 4.2 Data set generated by adding OWL file (without reasoner)

Through our tests above, we can see our application has reached most of our design requirements. There is an obvious bug, as shown in Table 4.2. When too many new queries (over 100) are generated after rewriting, because of the limitation of Amazon AWS API, the application will throw an exception. The reason for the exception is parameter length overflow. For every query, we need to input several parameters to let our application find relevant data files, so too many queries must lead to too many parameters. To solve this, we store all the input parameters in a file, and retrieve parameters needed when querying, so that we can avoid this limitation. In practice, too many queries also lead to low query efficiency, so it's quite necessary to reduce the number of queries. Another weakness of this application is: the application cannot rewrite query which has instance in it. The reason is the rewriting algorithm is for TBox (Ontology definition). In the next section, we will do improve the application.

4.4 Improvement

In this section I will improve the application. There are two main improvements: one is to reduce the number of queries after rewriting; the other is to make the application support rewriting queries which contain instances.

First I'll solve the problem that it doesn't support rewriting queries containing instances. The rewriting algorithm is based on Ontology definition, so when there is instance in a query, the algorithm can't recognize it and throws out exception. An example of query which contains instance is: Q(?0) <- Person(?0), memberOf(?0,<http://www.Department0.University0.edu>). If we can find the class which instance <http://www.Department0.University0.edu> belongs to, replacing the instance with its class name, rewriting is possible. Assume the class which instance <http://www.Department0.University0.edu> belongs to is Class1. We change the query into: Q(?0) <- Person(?0), Class1(?1), memberOf(?0,?1). Rewrite the query, and we will get a series of new queries which contain "Class1". The last thing needs to be done is to remove the item "Class1(?1)" in all the queries, and replace "?1" in other items with <http://www.Department0.University0.edu>. To our application, the name of an instance is always related to the name of the class it belongs to. Thus it's easy for us to find the class name of an instance. For example, <http://www.Department0.University0.edu> is the instance of class "Department", and <http://www.Department0.University0.edu/AssistantProfessor0> is the instance of class "AssistantProfessor". To other general applications, it's not so easy to find out the class name that an instance belongs to (the whole Ontology definition or some of the semantic data may be read in.). Here I won't discuss more about the general situation.

After implementing the method above, we test it with the 4th query of LUBM queries. The query is: Q(?1,?2,?3) <- Professor(?0), worksFor(?0,<http://www.Department0.University0.edu>), name(?0,?1), emailAddress(?0,?2), telephone(?0,?3), in which <http://www.Department0.University0.edu> is the instance of class "Department". First replace the instance with "?4", and the query is transformed as: Q(?1,?2,?3) <- Department(?4), Professor(?0), emailAddress(?0,?2), name(?0,?1), telephone(?0,?3), worksFor(?0,?4). After rewriting, the new queries are listed in Table 4.3.

Table 4.3 New Queries Generated from Revised Q4

Sequence	New Query
1	Q(?0,?1,?2) <- AssistantProfessor(?3), Department(?4), emailAddress(?3,?1), name(?3,?0), telephone(?3,?2), worksFor(?3,?4)
2	Q(?0,?1,?2) <- AssociateProfessor(?3), Department(?4), emailAddress(?3,?1), name(?3,?0), telephone(?3,?2), worksFor(?3,?4)
3	Q(?0,?1,?2) <- Chair(?3), emailAddress(?3,?1), name(?3,?0), telephone(?3,?2)
4	Q(?0,?1,?2) <- College(?3), Department(?4), emailAddress(?5,?1), headOf(?5,?3), name(?5,?0), telephone(?5,?2), worksFor(?5,?4)
5	Q(?0,?1,?2) <- Dean(?3), Department(?4), emailAddress(?3,?1), name(?3,?0), telephone(?3,?2), worksFor(?3,?4)
6	Q(?0,?1,?2) <- Department(?3), FullProfessor(?4), emailAddress(?4,?1), name(?4,?0), telephone(?4,?2), worksFor(?4,?3)
7	Q(?0,?1,?2) <- Department(?3), VisitingProfessor(?4), emailAddress(?4,?1), name(?4,?0), telephone(?4,?2), worksFor(?4,?3)
8	Q(?0,?1,?2) <- Department(?3), advisor(?4,?5), emailAddress(?5,?1), name(?5,?0), telephone(?5,?2), worksFor(?5,?3)
9	Q(?0,?1,?2) <- Department(?3), emailAddress(?4,?1), headOf(?4,?3), name(?4,?0), telephone(?4,?2)
10	Q(?0,?1,?2) <- Department(?3), emailAddress(?4,?1), name(?4,?0), telephone(?4,?2), tenured(?4,?5), worksFor(?4,?3)
11	Q(?1,?2,?3) <- Department(?4), Professor(?0), emailAddress(?0,?2), name(?0,?1), telephone(?0,?3), worksFor(?0,?4)

In the table we see the 3rd new query doesn't contain "?4", so we can't replace the instance back to the query. The example told us that the variable we added in the query may be abandoned after rewriting. If the variable is abandoned, then the instance can't affect the query results, so the query results can't be correct. In order to keep the variable in the query, we add it into the antecedent of the query. To the example above, we first transform it into:Q(?1,?2,?3,?4) <- Department(?4), Professor(?0), emailAddress(?0,?2), name(?0,?1), telephone(?0,?3), worksFor(?0,?4), and then rewrite it.

To reduce the number of new queries after rewriting, we can imagine, because all new queries are generated from the orginal query and the Ontology definition, as long as the rewriting algorithm is right, any deduction to the new queries is not correct for the Ontology definition, and may lead to imcomplete query results. Thus, in order to reduce the number of new queries, we have to seek for the help of ABox. Observing our data sets, we found that text files of many classes and properties do not exist, or have no data in them, if the data set has not been materialized. In practice, if a system uses query rewriting to retrieve semantic data, it's unnecessary to materialize to data set, since materialized data will take more space. Thus we can assume:

Without data materialization, it's very possible that in a data set, text files of many classes and properties don't exist or have no data in them.

Based on the assumption, we use following method to reduce the number of new queries generated.

For any query such as Q(?0) <- Person(?0), Department(?1), memberOf(?0,?1), the relation between each condition is "AND". If any of the conditions can't be satisfied (no data), the whold query will return no result. For example, for the query above, if the file "person.txt" corresponding to class "Person" doesn't have any data in it, there will be no result returned after the query execution. Hence, before executing a query, if we first check if each text file corresponding to the classes and properties in the query has data, we can decide beforehand whether to submit the query. If any of the text files doesn't exist or has no data (of 0 size), we can abandon the query.

After query rewriting, we read all data files of a data set on Amazon S3 with listObjects() method of AmazonS3 object. For each file object, if its size is not 0, we put its name in a list "nonemptyobjects". For each query in the generated queries, if any of its query condition is not in list "nonemptyobjects", the query will be abandoned.

Now test the improved application. We will still test data set of 10 universities. The test queries are the 11 queries except the 2nd, 9th, 14th query. All tests are with query rewriting. Test results are shown in Table 4.4.

Table 4.4 Test Results after Improvements

Query	**Containing Instance?**	**Number of New Queries Before**	**Number of New Queries After**	**Execution Result**
Q1	Yes	1	1	Correctly Finished
Q3	Yes	1	1	Correctly Finished
Q4	Yes	12	6	Correctly Finished
Q5	Yes	4	3	Correctly Finished
Q6	No	169	66	Correctly Finished
Q7	Yes	37	18	Correctly Finished
Q8	Yes	216	72	Correctly Finished
Q10	Yes	37	18	Correctly Finished
Q11	Yes	6	0	Correctly Finished
Q12	Yes	18	8	Correctly Finished
Q13	Yes	5	3	Correctly Finished

Chapter 5 Conclusion

Semantic Web is an extension to the current WWW. WWW is document oriented, while Semantic Web is data oriented. The aim of it is to change the way of information sharing of current Internet, which is based on text exchange. By describing semantic information using Ontology, Semantic Web provides intellegent and automated information sharing. In recent years, the research of Semantic Web has attracted many researchers, and from theory study to practical application, the results have been fruitful.

Our research focuses on reasoning and querying semantic data in distributed environment. As an extension to current WWW, Semantic Web is obviously applied in distributed environment. An Ontology usually contains knowledge of a domain, which will be used by many researchers or users. We can imagine, even one Ontology may contain a huge amount of data, and each application party (for example, in our LUBM case, each university) usually would keep its data independent. Thus research focusing on distributed environment is practial and necessary.

There are two main method to execute correctly semantic querying: one is to materialize semantic data with reasoner, and query materialized data with ordinary queries; the other is to reason input query with Ontology definition and rewrite it into a series of new queries, and the query result is the combination of the results of all new queries. Many researches have shown, considering the amount of semantic data, materialization is inefficient, and takes extra space, while few researches have compared the efficiency of these two methods. This thesis has conducted such comparison through a series of test, with the help of Amazon cloud service and LUBM's Ontology, data and test queries. The test results show that even compared with directly querying materialized data, the efficiency of querying after query rewriting is not low. Hence, in practical application, for unmaterialized semantic data, in order to get correct query result, query rewriting is obviously a better solution.

Based on the application, integrating previous work, we developed a semantic data querying application on Amazon platform and Hadoop framework. With this application, a user can input a query on local machine, query semantic data stored on cloud end, and get result and statistics on local machine. We also developed data management functions, including adding and deleting semantic data. The user can decide whether to materialize added data. By reimplement interfaces, our application can be transplanted to other platforms, such as independent Hadoop cluster.

By implementing this applicationm, we gained experience on developing similar applications. It also helps with researches in other aspects of Semantic Web. Our research also shows some semantic queries are very inefficient The reaction time may be as long as tens of minutes. It's unacceptable for users in practical application. In our future research, one the basis of correct execution, how to improve the query efficiency to large amount of semantic data in distributed environment will be an emphysis.

The future of Semantic Web is exciting, while its implementation and application is a complicated and huge project. More endeavor from researchers in Artificial Intelligence and Semantic Web area should be poured into it, so that Semantic Web can improve the way information is exchanged, and users will soon exeperice the power of it.

References

[1] Gruber, Thomas R. "A translation approach to portable ontology specifications". Knowledge Acquisition 5 (2): 199–220.

[2] Hans-Jörg Happel, Stefan Seedorf. Applications of Ontologies in Software Engineering. Universität Mannheim.

[3] Koen H. van Dam, James Keirstead. Re-use of an ontology for urban energy systems modelling. In Proceedings of the 3rd International Conference onf Infrastructure Systems and Services: Next Generation Infrastructure Systems for Eco-Cities (INFRA), 11--13 November 2010, Shenzhen, China.

[4] Eduard Hovy. Using an Ontology to Simplify Data Access. University of Southern California.

[5] Demner-Fushman D, Lin J. Answer Extraction, Semantic Clustering, and Extractive Summarization for Clinical Question Answering. Proceedings of the 21th International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL 2006), July 2006, Sydney, Australia.

[6] Werner Ceustersa, Barry Smithb, Jim Flanagana. Ontology and Medical Terminology: Why Description Logics Are Not Enough. Language and Computing nv., Zonnegem, Belgium.

[7] Köhler, J. and Schulze-Kremer, S. (2002). The Semantic Metadatabase (SEMEDA): Ontology Based Integration of Federated Molecular Biological Data Sources. In Silico Biology 2, 0021.

[8] Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., Harris, M. A., Hill, D. P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J. C., Richardson, J. E., Ringwald, M., Rubin, G. M. and Sherlock, G. (2000). Gene Ontology: tool for the unification of biology. The Gene Ontology Consortium. Nature Genet. 25, 25-29.

[9] L. Cambrésy, S. Derriere, P. Padovani. Ontology of Astronomical Object Types Use Cases.

[10] Cassidy, Patrick. Toward an open-source foundation ontology representing the Longman's defining vocabulary: The COSMO Ontology OWL version, in Proc. Third International Ontology for the Intelligence Community Conference, CEUR Workshop Proceedings, vol. 440, Fairfax, VA.

[11] Nadia Anwar, Gary Bader, etc. BioPAX-Biological Pathways Exchange Language.

[12] Lomax J. Get ready to GO! A biologist's guide to the Gene Ontology. Brief. Bioinformatics. Sep 2005;6(3):298-304.

[13] Bentley White. ISO 15926-Lifecycle Data for Process Plant.

[14] Qiming Fang, Ying Zhao, Guangwen Yang, and Weimin Zheng. Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning. Tsinghua National Laboratory for Information Science and Technology Department of Computer Science and Technology, Tsinghua University.

[15] Aimilia Magkanaraki, Grigoris Karvounarakis, Ta Tuan Anh, Vassilis Christophides, Dimitris Plexousakis. ONTOLOGY STORAGE AND QUERYING.

[16] Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. Scientific American. [2006-10-5].

[17] Grigoris Antoniou, F. v. H. (2008). A Semantic Web Primer.

[18] W3C XML Schema. http://www.w3.org/XML/Schema.

[19] Brian McBride. RDF Primer. Hewlett-Packard Laboratories.

[20] Deborah L. McGuinness, Frank van Harmelen. OWL Web Ontology Language Overview.

[21] Xiaofeng Song, Fagen Tang. Reasoning Technologies based on Ontology. Computer School, Beijing University of Aeronautics.

[22] Ian Horrocks. Description Logic Reasoning. University of Manchester Manchester, UK.

[23] Zhiyong Wang. Research and Implementation of Ontology Querying and Reasoning. Master thesis in Central South University, China.

[24] Gruman, Galen. What cloud computing really means. InfoWorld. Retrieved 2009-06-02.

[25] Biddick, Michael. Why You Need A SaaS Strategy. Retrieved 2010-03-10.

[26] Jack Schofield. Google angles for business users with 'platform as a service'.

[27] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, etc. PNUTS: Yahoo!'s Hosted Data Serving Platform. Yahoo! Research.

[28] http://cloud.saaser.cn/cloudnews/20100226/3871.html.

[29] http://www.microsoft.com/windowsazure/.

[30] Naveen Ashish, Craig A. Knoblock, Cyrus Shahabi. Optimizing Information Agents by Selectively Materializing Data. University of Southern California.

[31] Joseph Fong, Ringo Pang, Anthony Fong, Francis Pang, Kenny Poon. Concurrent data materialization for Object-Relational database with semantic metadata. City University of Hong Kong.

[32] Maria Esther Vidal, Louiqa Raschid, Natalia Marquez, Marelis Cardenas, Yao Wu. Query Rewriting in the Semantic Web. Universidad Simon Bolivar, University of Maryland.

[33] Alexandre Riazanov, UNB Saint John. Expressive Querying of Semantic Databases with Incremental Query Rewriting. Marcelo A. T. Aragao, Manchester University, Central Bank of Brazil.

[34] Roger Castillo, Christian Rothe, Ulf Leser. RDFMatView: Indexing RDF Data using Materialized SPARQL queries. Humboldt University of Berlin.

[35] Chulki Lee, Sungchan Park, Dongjoo Lee, Jae-won Lee, Ok-Ran Jeong, Sang-goo Lee. A Comparison of Ontology Reasoning SystemsUsing Query Sequences. Seoul National University, University of Illinois at Urbana-Champaign.

[36] http://hadoop.apache.org/.
[37] Yuan Yu, Pradeep Kumar Gunda, Michael Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. Microsoft Research.
[38] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System. Google, Inc.
[39] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Google, Inc.
[40] Fay Chang, Jeffrey Dean, Sanjay Ghemawat. Bigtable: A Distributed Storage System for Structured Data. Google, Inc.
[41] Tomasz Wiktor Wlodarczyk, Yi Han, Chunming Rong. Performance Analysis of Hadoop for Query Processing. University of Stavanger.
[42] Daniel J. Abadi, Adam Marcus. Scalable Semantic Web Data Management using Vertical Partitioning.
[43] Héctor Pérez-urbina, Boris Motik, Ian Horrocks. A Comparison of Query Rewriting Techniques for DL-Lite.
[44] Héctor Pérez-urbina, Boris Motik, Ian Horrocks. Rewriting Conjunctive Queries under Description Logic Constraints.
[45] Héctor Pérez-urbina, Ian Horrocks, Boris Motik. Practical Aspects of Query Rewriting for OWL 2.

Appendices

1. LUBM Test Queries (datalog format)

Q1:

```
SELECT ?0
WHERE { ?0 rdf:type :GraduateStudent .
        ?0 :takesCourse <http://www.Department0.University0.edu/GraduateCourse0> }
```

Q2:

```
SELECT ?0 ?1
WHERE { ?0 rdf:type :GraduateStudent .
        ?1 rdf:type :University .
        ?2 rdf:type :Department .
        ?0 :memberOf ?2 .
        ?2 :subOrganizationOf ?1 .
        ?0 :undergraduateDegreeFrom ?1 }
```

Q3:

```
SELECT ?0
WHERE { ?0 rdf:type :Publication .
        ?0 :publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0> }
```

Q4:

```
SELECT ?1 ?2 ?3
WHERE { ?0 rdf:type :Professor .
        ?0 :worksFor <http://www.Department0.University0.edu> .
        ?0 :name ?1 .
        ?0 :emailAddress ?2 .
        ?0 :telephone ?3 }
```

Q5:

```
SELECT ?0
WHERE { ?0 rdf:type :Person .
```

```
        ?0 :memberOf <http://www.Department0.University0.edu> }
```

Q6:

```
SELECT ?0
WHERE { ?0 rdf:type :Student }
```

Q7:

```
SELECT ?0 ?1
WHERE { ?0 rdf:type :Student .
        ?1 rdf:type :Course .
        <http://www.Department0.University0.edu/AssociateProfessor0> :teacherOf ?1 .
        ?0 :takesCourse ?1 }
```

Q8:

```
SELECT ?0 ?2
WHERE { ?0 rdf:type :Student .
        ?1 rdf:type :Department .
        ?0 :memberOf ?1 .
        ?1 :subOrganizationOf <http://www.University0.edu> .
        ?0 :emailAddress ?2 }
```

Q9:

```
SELECT ?0 ?1 ?2
WHERE { ?0 rdf:type :Student .
        ?1 rdf:type :Faculty .
        ?2 rdf:type :Course .
        ?0 :advisor ?1 .
        ?0 :takesCourse ?2 .
        ?1 :teacherOf ?2 }
```

Q10:

```
SELECT ?0
WHERE { ?0 rdf:type :Student .
        ?0 :takesCourse <http://www.Department0.University0.edu/GraduateCours> }
```

```
Q11:
SELECT ?0
WHERE { ?0 rdf:type :ReserchGroup .
        ?0 :subOrganizationOf <http://www.University0.edu> }

Q12:
SELECT ?0 ?1
WHERE { ?0 rdf:type :Chair .
        ?1 rdf:type :Department .
        ?0 :worksFor ?1 .
        ?1 :subOrganizationOf <http://www.University0.edu> }

Q13:
SELECT ?0
WHERE { ?0 rdf:type :Person .
        <http://www.University0.edu> :hasAlumnus ?0 }

Q14:
SELECT ?0
WHERE { ?0 rdf:type :UndergraduateStudent }
```

2. Queries Generated from Query 2 by Requiem (SPARQL format)

```
Q2-0:
SELECT ?0 ?1
WHERE { ?2 rdf:type :Department . ?0
rdf:type :GraduateStudent . ?0 :headOf ?2 . ?2 :subOrganizationOf ?1 . ?0 :undergraduateDegreeFro
m ?1 }
Q2-1:
SELECT ?0 ?1
WHERE { ?2 rdf:type :Department . ?0
rdf:type :GraduateStudent . ?2 :member ?0 . ?2 :subOrganizationOf ?1 . ?0 :undergraduateDegreeFr
om ?1 }
Q2-2:
SELECT ?0 ?1
```

WHERE { ?2 rdf:type :Department . ?0
rdf:type :GraduateStudent . ?0 :memberOf ?2 . ?2 :subOrganizationOf ?1 . ?0 :undergraduateDegree
From ?1 }

Q2-3:

SELECT ?0 ?1
WHERE { ?2 rdf:type :Department . ?0
rdf:type :GraduateStudent . ?2 :subOrganizationOf ?1 . ?0 :undergraduateDegreeFrom ?1 . ?0 :work
sFor ?2 }

3. Clojure File Corresponding to Query 2

```
(ns job.q2
   (:use cascalog.api)
   (:require [cascalog [vars :as v] [workflow :as w] [ops :as c]])
   (:gen-class)
)

(import 'java.lang.System)
(import 'java.util.Date)

(defn getTimeStamp []
   (str (. System currentTimeMillis) " " (new Date)))

(defn copyFile [source hdfsTarget]
   (let [sourceTable (hfs-textline source)]
      (?<- (hfs-textline hdfsTarget) [?x] (sourceTable ?x))
   )
)

(defn generateHdfsPath [sPath]
   (.replaceFirst sPath "s3n://" "hdfs:///")
)

(defn textline-parsed [dir num-fields]
   (let [outargs (v/gen-nullable-vars num-fields)
         source (hfs-textline dir)]
```

```
    (<- outargs (source ?line) (c/re-parse [#"[^\s]+"] ?line :>> outargs) (:distinct false))
  )
)

(defn GraduateStudent-data [dir]
  (textline-parsed dir 1)
)
(defn University-data [dir]
  (textline-parsed dir 1)
)
(defn Department-data [dir]
  (textline-parsed dir 1)
)
(defn memberOf-data [dir]
  (textline-parsed dir 2)
)
(defn subOrganizationOf-data [dir]
  (textline-parsed dir 2)
)
(defn undergraduateDegreeFrom-data [dir]
  (textline-parsed dir 2)
)

; query function
(defn q2 [GraduateStudent_source University_source Department_source memberOf_source
subOrganizationOf_source undergraduateDegreeFrom_source  dest]
  (let [GraduateStudent (GraduateStudent-data GraduateStudent_source)
University (University-data University_source)
Department (Department-data Department_source)
memberOf (memberOf-data memberOf_source)
subOrganizationOf (subOrganizationOf-data subOrganizationOf_source)
undergraduateDegreeFrom (undergraduateDegreeFrom-data undergraduateDegreeFrom_source)
]
     (?<- (hfs-textline dest) [?0 ?1] (GraduateStudent ?1) (University ?1) (Department ?2)
(memberOf ?0 ?2) (subOrganizationOf ?2 ?1) (undergraduateDegreeFrom ?0 ?1) )
```

```
  )
)

(defn -main [GraduateStudent_source University_source Department_source memberOf_source
subOrganizationOf_source undergraduateDegreeFrom_source dest1 ]
  (let []
      (. (. System out) println (str (getTimeStamp) " - " "-main start"))
      (. (. System out) println (str (getTimeStamp) " - " "query start"))
      (q2 GraduateStudent_source University_source Department_source memberOf_source
subOrganizationOf_source undergraduateDegreeFrom_source dest1)
      (. (. System out) println (str (getTimeStamp) " - " "query end"))
      (. (. System out) println (str (getTimeStamp) " - " "-main end"))
  )
)
```