Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools

Mohammad Farhan Husain*, Latifur Khan†, Murat Kantarcioglu‡ and Bhavani Thuraisingham§
Department of Computer Science
University of Texas at Dallas
800 West Campbell Road, Richardson, TX 75080-3021
*mfh062000@utdallas.edu
†lkhan@utdallas.edu
‡muratk@utdallas.edu
§bhavani.thuraisingham@utdallas.edu

***Abstract*—Cloud computing is the newest paradigm in the IT world and hence the focus of new research. Companies hosting cloud computing services face the challenge of handling data intensive applications. Semantic web technologies can be an ideal candidate to be used together with cloud computing tools to provide a solution. These technologies have been standardized by the World Wide Web Consortium (W3C). One such standard is the Resource Description Framework (RDF). With the explosion of semantic web technologies, large RDF graphs are common place. Current frameworks do not scale for large RDF graphs. In this paper, we describe a framework that we built using Hadoop, a popular open source framework for Cloud Computing, to store and retrieve large numbers of RDF triples. We describe a scheme to store RDF data in Hadoop Distributed File System. We present an algorithm to generate the best possible query plan to answer a SPARQL Protocol and RDF Query Language (SPARQL) query based on a cost model. We use Hadoop's MapReduce framework to answer the queries. Our results show that we can store large RDF graphs in Hadoop clusters built with cheap commodity class hardware. Furthermore, we show that our framework is scalable and efficient and can easily handle billions of RDF triples, unlike traditional approaches.**

***Keywords*-RDF; Hadoop; Cloud; Semantic Web;**

I. Introduction

Cloud computing is now the center of attraction for large enterprises looking for ways to be more cost efficient. A lot of research is going on in this arena to make cloud computing more efficient, secure and affordable. Semantic web is an evolving technology which can be utilized for this purpose. Semantic Web technologies are being developed to present data in a more efficient way so that such data can be retrieved and understood by both human and machine. At present, web pages are published in plain html files which are not suitable for reasoning. Instead, the machine treats these html files as a bag of keywords. Researchers are developing Semantic Web technologies that have been standardized to address such inadequacies. The most prominent standards are Resource Description Framework[1] (RDF) and SPARQL Protocol and RDF Query Language[2] (SPARQL). RDF is the standard for storing and representing data and SPARQL is a query language to retrieve data from an RDF store. The power of these Semantic Web technologies can be successfully harnessed in cloud Computing environment to provide the user with capability to efficiently store and retrieve data for data intensive applications. Synergy between the semantic web and cloud computing fields offers great benefits, such as standards for data representation across frameworks.

The need for cloud computing hosting companies to handle data intensive applications scalably a major issue. Even with huge amounts of data, data intensive systems should not be bogged down and their performance should not deteriorate. Designing such scalable systems is not a trivial task. When it comes to semantic web data such as RDF, we face similar challenges. With storage becoming cheaper and the need to store and retrieve large amounts of data, developing systems to handle trillions of RDF triples requiring tera/peta bytes of disk space is no longer a distant prospect. Researchers are already working on billions of triples [16], [19]. Competitions are being organized to encourage researchers to build efficient repositories[3]. At present, there are just a few frameworks (e.g. Jena[4], Sesame[5], BigOWLIM[6]) for Semantic Web technologies, and these frameworks are not scalable for large RDF graphs. Jena Semantic Web Framework is one of the most popular ones which has several models to store data: in-memory model, SDB model[7], TDB model[8], etc. They are all designed for a single machine scenario; hence, they are not scalable when it comes to terabytes of data. Only 10 million triples can be processed in a Jena in-memory model running on a machine having 2 GB of main memory. Another such framework

[1] http://www.w3.org/TR/rdf-primer

[2] http://www.w3.org/TR/rdf-sparql-query
[3] http://challenge.semanticweb.org
[4] http://jena.sourceforge.net
[5] http://www.openrdf.org
[6] http://www.ontotext.com/owlim/big/index.html
[7] http://jena.hpl.hp.com/wiki/SDB
[8] http://jena.hpl.hp.com/wiki/TDB

978-0-7695-4130-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CLOUD.2010.36

is Sesame which has a native storage. It has the same limitations as Jena: it cannot handle arbitrarily large datasets. Therefore, storing a large number of RDF triples and efficiently querying them is a challenging problem which has yet to be solved. BigOWLIM has both an in-memory and persistent storage. However, as it works as a lower layer in Sesame API stack it inherits all the disadvantages of Sesame.

As a solution, a distributed system can be built to overcome the scalability and performance problems of current Semantic Web frameworks. Databases are being distributed in order to provide such scalable solutions. However, to date, there is no distributed repository for storing and managing RDF data. Researchers have only recently begun to explore the problems and technical solutions which must be addressed in order to build such a distributed system for semantic web data. One promising line of investigation involves making use of readily available distributed database systems or relational databases. Such database systems can use relational schema for the storage of RDF data. Optimal relational schemas are being probed for this purpose [2]. There is also the possibility of constructing a distributed system from scratch. Here, there will be an opportunity to design and optimize a system with specific application to RDF data. With the former approach, performance and scalability will be issues due to the fact that the systems were developed for relational data. In the latter approach, we would be reinventing the wheel. Hence, instead of starting with a blank slate, it should be possible to begin with a Cloud Computing framework or a generic distributed storage system which can be used in a Cloud Computing platform and tailor it to meet the needs of semantic web data. A semantic web repository could then be built on top of that.

Companies pioneering cloud computing such as Salesforce.com and Amazon have platforms such as EC2[9], S3[10], Force.com[11] etc. These are proprietary, closed source platforms. However, Hadoop[12] is an emerging Cloud Computing tool which is open source and supported by Amazon, the leading Cloud Computing hosting company. It is a distributed file system where files can be saved with replication, and would be an ideal candidate for building a storage system. Hadoop features high fault tolerance and great reliability. In addition, it also contains an implementation of the MapReduce [6] programming model, a functional programming model which is suitable for the parallel processing of large amounts of data. By partitioning data into a number of independent chunks, MapReduce processes run on these chunks, making parallelization easier.

In this paper, we will describe our work with RDF data in which we introduce a schema to store RDF data in Hadoop. In the preprocessing stage, we process RDF data and put it in files in the distributed file system according to our schema. We have chosen two benchmark datasets for our experiments, and we have a retrieval mechanism using MapReduce programming. All our queries are executed against the two benchmark datasets on different sizes of data sets ranging from 100 million triples to 1 billion triples. Our goal is to answer SPARQL queries as efficiently as possible using summary statistics about the data. In Hadoop, the unit of computation is called a MapReduce job (later referred to simply as job). A user submits a job to the Hadoop job tracker which is responsible for running the job. We find that for many queries, one job is not enough. This is because in Hadoop the jobs we execute to perform joins cannot communicate with each other, hence joins dependent on other joins cannot be executed in the same job. Therefore, we need an algorithm to determine how many jobs are required for a given query. We have devised such an algorithm, which uses backtracking combined with a two coloring scheme to generate multiple plans. It then chooses the best plan based on a cost model. The plan not only determines the number of jobs, but also their sequence and inputs.

Our contributions are as follows: first, we design a storage scheme to store RDF data in plain text files in Hadoop distributed file system (HDFS[13]). Second, we devise an algorithm which, based on a cost model, determines the best query plan needed to answer a SPARQL query. Third, we build a cost model for query processing plans. Finally, we demonstrate that our approach performs better than Jena, the most widely used semantic web framework.

The remainder of this paper is organized as follows: in Section II, we discuss relevant research. In Section III, we discuss the system architecture and our data storage scheme. In Section IV, we discuss how we answer a SPARQL query. In Section V, we present the results of our experiments. Finally, in Section VI, we draw some conclusions and discuss probable areas which we have identified for improvement in the future.

II. Related Works

MapReduce is currently an evolving technology. This technology has been well received by the community which handles large amounts of data. Researchers and enterprises are using MapReduce technology in fields such as web indexing for search, data mining and semantic web. With regard to the domain, first, we describe works with MapReduce in web search and data mining areas and next we will discuss works related to semantic web.

Google uses MapReduce for web indexing, data storage and social networking [4]. Yahoo! uses MapReduce extensively in their data analysis tasks [17]. IBM has successfully experimented with a scale-up scale-out search framework using MapReduce technology [13].

[9]http://aws.amazon.com/ec2
[10]http://aws.amazon.com/s3
[11]http://www.salesforce.com/platform
[12]http://hadoop.apache.org

[13]http://hadoop.apache.org/core/docs/r0.18.3/hdfs_design.html

Researchers have used MapReduce to scale up classifiers for mining petabytes of data [15]. They have worked on data distribution and partitioning for data mining, and have applied three data mining algorithms to test the performance. Data mining algorithms are being rewritten in different forms to take advantage of MapReduce technology. In [5], researchers rewrite well-known machine learning algorithms to take advantage of multicore machines by leveraging MapReduce programming paradigm. Another area where this technology is successfully being used is simulation [12]. In [3], researchers reported an interesting idea of combining MapReduce with existing relational database techniques. These works differ from our research in that we use MapReduce for semantic web technologies. Researchers are also exploring other ways to build framework for semantic web data. In [1] and [2], researchers reported a vertically partitioned DBMS for storage and retrieval of RDF data. They observed performance improvement with their scheme over traditional relational database schemes. Researchers have also used MapReduce for inferencing RDF data. In [22], they have proposed a solution to do scalable distributed reasoning of RDF data using MapReduce. The solution proposed in [23] can also be implemented by MapReduce.

In the semantic web arena, there has not been much work done with MapReduce technology. The closest match to what we have done is the BioMANTA[14] project. Researchers of that project have done some work regarding large RDF data storage in Hadoop. They propose extensions to RDF Molecules [7] and implement a MapReduce based Molecule store [16]. They use MapReduce to answer the queries. They have queried a maximum of 4 million triples. Our work differs in the following ways: first, we have queried 1 billion triples. Second, we have devised a storage schema which is tailored to improve query execution performance for RDF data. We store RDF triples in files based on the predicate of the triple and the type of the object. Finally, we also have an algorithm to determine the best query processing plan to answer a query based on a cost model which uses the summary statistics. By the plan, we can determine the input files of a job and the order in which they should be run. To the best of our knowledge, we are the first ones to come up with a storage schema for RDF data using flat files in HDFS, and a MapReduce job determination algorithm to answer a SPARQL query.

At present, BigOWLIM is the fastest and most scalable semantic web framework available. However, it is not as scalable as our framework and requires very high end and costly machines. It requires a lot of main memory to load large datasets and it has a long loading time. As our experiments showed (Please see Section V-A), it does not perform well when there is no bound object in a query. However, the performance of our framework is not affected in such a case.

[14]http://www.itee.uq.edu.au/ eresearch/projects/biomanta

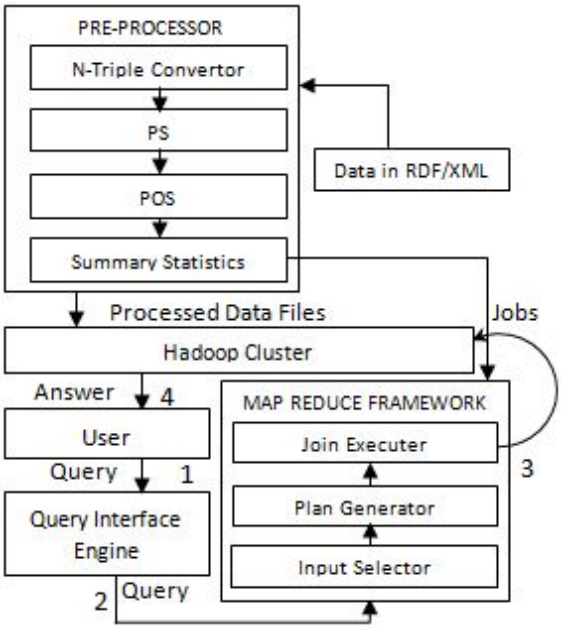

Figure 1. The System Architecture

In our previous work [11], we have proposed a greedy algorithm to generate a query processing plan. But that algorithm has two limitations. First, it uses a cost model which only considers total number of Hadoop jobs as the cost of a query plan. Next, it does not always generate the best query plan according to that fixed cost model.

III. Proposed Architecture

Our architecture consists of two components. The upper part of Figure 1 depicts the data preprocessing component, and the lower part shows the one which answers a query.

We have three subcomponents for data generation and preprocessing. We convert RDF/XML[15] to N-Triples[16] serialization format using our N-Triples Converter component. The PS component takes the N-Triples data and splits it into predicate files. The predicate based files are then fed into the POS component which splits the predicate files into smaller files based on the type of objects. These steps are described in Section III-B, III-C and III-D.

Our MapReduce framework has three sub-components in it. It takes the SPARQL query from the user and passes it to the Input Selector (see Section IV-A) and Plan Generator. This component selects the input files, by using our algorithm described in Section IV-C, decides how many MapReduce jobs are needed and passes the information to the Join Executer component which runs the jobs using MapReduce framework. It then relays the query answer from Hadoop to the user.

[15]http://www.w3.org/TR/rdf-syntax-grammar
[16]http://www.w3.org/2001/sw/RDFCore/ntriples

A. Data Generation and Storage

For our experiments, we use the LUBM [8] dataset. It is a benchmark datasets designed to enable researchers to evaluate their semantic web repository performances [10]. The LUBM data generator generates data in RDF/XML serialization format. This format is not suitable for our purpose because we store data in HDFS as flat files and so to retrieve even a single triple we need to parse the entire file. Therefore we convert the data to N-Triples to store the data, because with that format we have a complete RDF triple (Subject, Predicate and Object) in one line of a file, which is very convenient to use with MapReduce jobs. The processing steps to go through to get the data in our intended format are described in following sections.

B. File Organization

We do not store the data in a single file because, in Hadoop and MapReduce Framework, a file is the smallest unit of input to a MapReduce job and in absence of caching, a file is always read from the disk. If we have all the data in one file, the whole file will be input to jobs for each query. Instead, we divide the data into multiple smaller files. The splitting is done in two steps which we discuss in the following sections.

C. Predicate Split (PS)

In the first step, we divide the data according to the predicates. In real world RDF datasets, the number of distinct predicates is no more than 10 or 20 [21]. This division immediately enables us to cut down the search space for any SPARQL query which does not have a variable[17] predicate. For such a query, we can just pick a file for each predicate and run the query on those files only. For simplicity, we name the files with predicates, e.g. all the triples containing a predicate *p1:pred* go into a file named *p1-pred*. However, in case we have a variable predicate in a triple pattern[18] and if we cannot determine the type of the object, we have to consider all files. If we can determine the type of the object then we consider all files having that type of object. We discuss more on this in Section IV-A.

D. Predicate Object Split (POS)

1) Split Using Explicit Type Information of Object: In the next step, we work with the explicit type information in the *rdf_type* file. The file is first divided into as many files as the number of distinct objects the *rdf:type* predicate has. For example, if in the ontology the leaves of the class hierarchy are $c_1, c_2, ..., c_n$ then we will create files for each of these leaves and the file names will be like *rdf-type_*c_1, *rdf-type_*c_2, ... , *rdf-type_*c_n. Please note that the object values $c_1, c_2, ..., c_n$ are no longer needed to be stored inside the file as they can be easily retrieved from the file name. This further reduces the amount of space needed to store the data. We generate such a file for each distinct object value of the predicate *rdf:type*.

2) Split Using Implicit Type Information of Object: We divide the remaining predicate files according to the type of the objects. Not all the objects are URIs, some are literals. The literals remain in the file named by the predicate: no further processing is required for them. The type information of a URI object is not mentioned in these files but they can be retrieved from the *rdf-type_** files. The URI objects move into their respective file named as *predicate_type*. For example, if a triple has the predicate *p* and the type of the URI object is c_i, then the subject and object appears in one line in the file *p_*c_i. To do this division we need to join a predicate file with the *rdf-type_** files to retrieve the type information.

In Table I we show the number of files we get after *PS* and *POS* steps. We can see that eventually we organize the data into 41 files.

Step	Files	Size (GB)	Space Gain
N-Triples	20020	24	-
PS	17	7.1	70.42%
POS	41	6.6	7.04%

Table I
DATA SIZE AT VARIOUS STEPS FOR 1000 UNIVERSITIES

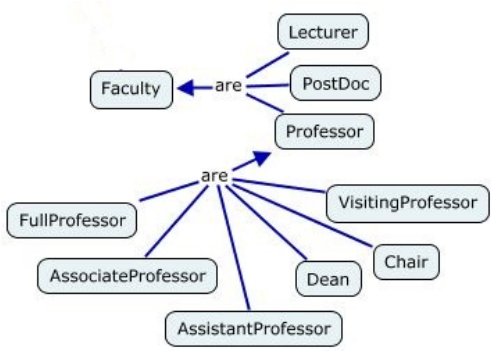

Figure 2. Partial LUBM Ontology

Table I shows the number of files and size gain we get at each step for data of 1000 universities. LUBM generator generates 20020 small files of total 24 GB size. After splitting the data according to predicates the size drastically comes down to only 7.1 GB which is a 70.42% gain. This happens because of the absence of predicate columns and also the prefix substitution. At this step, we only get 17 files as there are 17 unique predicates in the LUBM data set. In the final step, we again gain 7.04% space as the split *rdf-type* files no longer has the object column. The number of files increases to 41 as predicate files are split according to the type of the objects.

It should be noted that if a SPARQL query specifies type information of a variable which is not a leaf in the

[17]http://www.w3.org/TR/rdf-sparql-query/#sparqlQueryVariables
[18]http://www.w3.org/TR/rdf-sparql-query/#sparqlTriplePatterns

Table II
SAMPLE DATA FOR LUBM QUERY 9

type_Student	ub:advisor		ub:takesCourse		ub:teacherOf	
GS1	GS2	A2	GS1	C2	A1	C1
GS2	GS1	A1	GS3	C1	A2	C2
GS3	GS3	A3	GS3	C3	A3	C3
GS4	GS4	A4	GS2	C4	A4	C4
			GS1	C1	A5	C5
			GS4	C2		

ontology tree, we consider all the files having leaf types of the subtree rooted at that type node. For example, LUBM Query 4, shown in Listing 2, specifies type of the variable as *Professor* which is not a leaf in the LUBM ontology[19], part of which is shown in Figure 2. For this query, we input files *type_AssistantProfessor*, *type_AssociateProfessor*, *type_FullProfessor* and *type_VisitingProfessor* as these are the leaves of the subtree rooted at node *Professor*. As we do not add inferred triples to the dataset in our framework, we do not have files *type_Dean* and *type_Chair* as in LUBM dataset there is no explicit *Dean* and *Chair*.

E. Example Data

In Table II we have shown sample data for three predicates. The left most column shows the type file for $student$ objects after the splitting by using explicit type information in *POS* step. It lists only the subjects of the triples having *rdf:type* predicate and $student$ object. The rest of the columns show the the $advisor$, $takesCourse$ and $teacherOf$ predicate files after the splitting by using implicit type information in *POS* step. The prefix *ub:* stands for *http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#*. Each row has a pair of subject and object. In all cases, the predicate can be retrieved from the filename.

IV. MAPREDUCE FRAMEWORK

Our MapReduce Framework is where we answer queries. The challenges we meet to solve a SPARQL query are as follows: first, we must determine the number of jobs needed to answer a query. Second, we need to minimize the size of intermediate files so that data copying and network data transfer is reduced. Finally, we must determine number of reducers. We run one or more MapReduce jobs to answer one query. We use the map phase for selection and the reduce phase for join.

To answer a SPARQL query using the MapReduce framework, we often require more than one job. The reason is that, most of the time, more than one join is required to answer the queries presented. One job is not sufficient to perform all the joins as the MapReduce processes in Hadoop have no inter-process communication. Hence, processing a piece of data cannot be dependent on the outcome of any other processing, which is essential for joins. This is why we may need more than one job to answer a query. Each job may depend on the output of the previous job, if there is any.

[19]http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl

Listing 1. LUBM Query 2

```
1 SELECT ?X, ?Y, ?Z WHERE {
2 ?X rdf:type ub:GraduateStudent .
3 ?Y rdf:type ub:University .
4 ?Z rdf:type ub:Department .
5 ?X ub:memberOf ?Z .
6 ?Z ub:subOrganizationOf ?Y .
7 ?X ub:undergraduateDegreeFrom ?Y }
```

Listing 2. LUBM Query 4

```
SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
?X rdf:type ub:Professor .
?X ub:worksFor <http://www.D0.U0.edu> .
?X ub:name ?Y1 .
?X ub:emailAddress ?Y2 .
?X ub:telephone ?Y3 }
```

A. Input Files Selection

Before determining the jobs, we select the files that need to be inputted to the jobs. We take the query submitted by the user and iterate over the triple patterns. In a triple pattern, if the both the predicate and object are variables and the object has not type information available or if the object is concrete, we select all the files in the dataset as input to the jobs and terminate the iteration. If the predicate is a variable but the object has type information available, we select all predicate files having object of that type and add them to the input file set. If the predicate is concrete but the object is variable without any type information, we add all files for the predicate to the input set. If the query has the type information of the object, we add the predicate file which has objects of that type to the input set. If a type associated with a predicate is not a leaf in the ontology tree, we add all subclasses which are leaves in the subtree rooted at the type node in the ontology. For example for Query 2 shown in Listing 1, for the first three triple patterns, we add the files *type_GraduateStudent*, *type_University* and *type_Department* to the input file set. For the fourth triple pattern (?X ub:memberOf ?Z), we have type information of the variable *?Z*. So we add the file *memberOf_Department* to the input file set. Similarly, we add files *subOrganizationOf_Department* and *undergraduateDegreeFrom_Department* to the input file set for the last two triple patterns. Another example shows the role of ontology in input file selection. In Query 4, shown in Listing 2, the first triple pattern has a concrete object *Professor*. But it is not a leaf in the ontology shown in Figure 2. So we add all the leaves rooted at the subtree at node *Professor* i.e. we add the files *type_AssistantProfessor*, *type_AssociateProfessor*, *type_FullProfessor* and *type_VisitingProfessor* to the input file list.

B. Cost Estimation for Query Processing

We run Hadoop jobs to answer a SPARQL query. In this section we discuss how we estimate cost of a job. But before doing that we introduce some definitions which we use later:

Definition 1 (Conflicting Joins, (*CJ*)). *Conflicting Joins is a pair of joins on different variables sharing a triple pattern.*

Definition 2 (NonConflicting Joins, *NCJ*). *Non-conflicting Joins is a pair of joins either not sharing any triple pattern or sharing a triple pattern and the joins are on same variable.*

Listing 3. LUBM Query 12

```
SELECT ?X WHERE {
?X rdf:type ub:Chair .
?Y rdf:type ub:Department .
?X ub:worksFor ?Y .
?Y ub:subOrganizationOf <http://www.U0.edu>}
```

An example illustrates the terms better. In Listing 3, we show LUBM Query 12. Lines 2, 3, 4 and 5 each has a triple pattern. If we do two joins, one between triple patterns in lines 2 and 4 on variable $?X$ and the other between triple patterns in lines 4 and 5 on variable $?Y$, they will be a CJ as triple pattern in line 4 takes part in two joins on two different variables. A NCJ would be one join between triple patterns in lines 2 and 4 on variable $?X$ and another join between triple patterns in lines 3 and 5 on variable $?Y$.

To answer a SPARQL query we may need more than one job. Therefore, cost estimation for processing a query requires individual cost estimation of each job that is needed to answer that query. A job contains three main tasks, which are reading, sorting and writing. We estimate the cost of a job based on these three tasks. For each task, a unit cost is assigned to each triple pattern it deals with. In the current model, we assume that costs for reading and writing are the same.

$$TotalCost = (\sum_{i=1}^{n-1} MI_i + MO_i + RI_i + RO_i) + MI_n + MO_n + RI_n \quad (1)$$

$$= (\sum_{i=1}^{n-1} job_i) + MI_n + MO_n + RI_n \quad (2)$$

$$job_i = MI_i + MO_i + RI_i + RO_i \quad (if \quad i < n) \quad (3)$$

Where, MI_i = Cost of Map Input phase for job i; MO_i = Cost of Map Output phase for job i; RI_i = Cost of Reduce Input phase for job i; RO_i = Cost of Reduce Output phase for job i.

Equation 1 is the total cost of processing a query. It is the summation of the individual costs of each job and selected phases of the final job. A job essentially performs a MapReduce task on the file data. Equation 2 shows the division of the MapReduce task into subtasks. Hence, to estimate the cost of each job, we will combine the estimated cost of each subtask.

Map Input (MI) phase: This phase reads the triple patterns from the selected input files stored in the HDFS. Therefore, we estimate the cost for the MI phase to be equal to the total number of triple patterns in each of the selected file.

Map Output Phase (MO): The estimation of the MO phase depends on the type of query being processed. If the query has no bound variable (e.g. [?X ub:worksFor ?Y]), then the output of the Map phase is equal to the input. All of the triple patterns are transformed into key-value pairs and given as output. Therefore, for such a query the MO cost will be the same as MI cost. However, if the query involves a bound variable, (e.g. [?Y ub:subOrganizationOf <http://www.U0.edu>]), then, before making the key-value pairs, the bound component selectivity estimation is applied. We get the estimation from the summary statistics we gather in the data preprocessing step. The resulting estimate for the triple patterns will account for the cost of Map Output phase. The selected triples are written to a local disk.

Reduce Input Phase (RI): In this phase, the triples from the Map output phase are read via HTTP and then sorted based on their key values. After sorting, the triples with identical keys are grouped together. Therefore, the cost estimation for the RI phase is equal to the MO phase. The number of key-value pairs that are sorted in RI is equal to the number of key-value pairs generated in the MO phase.

Reduce Output Phase (RO): The RO phase deals with performing the joins. Therefore, it is in this phase we use the join triple pattern selectivity summary statistics. However, once the triple pattern selectivity is used to estimate the triple patterns chosen in RO, it cannot be used again for the intermediate jobs. We lack the precise knowledge of the number of triple patterns selected after applying the join in the first job. Therefore, for the intermediate jobs, we take an upper bound on the Reduce Output and assume that the same number of triples will be handled in the RO phase.

Equation 3 shows a very important postulation. It illustrates the total cost of an intermediate job, when $i < n$, includes the cost of the RO phase in calculating the total cost of the job.

However, for the last job, when $i = n$, we do not consider the Reduce Output phase cost. The RO phase of the very last job of the query processing gives the final result for that query. Moreover, the final result of the query will be equal for all query plans. This implies that the cost of this phase will be equal for all the query plans. Therefore, we can ignore this cost for the final RO phase of the query processing.

C. Query Plan Generation

1) Motivation: As we discussed earlier, there may be more than one job necessary to answer a query. Hence, there

may be multiple ways a SPARQL query can be answered. These ways are typically called query plans. Each of the possible plans to answer a query has different performance in terms of time and space. So, there is a need to find the plan which would provide the best performance. An example can illustrate the issue better. Listing 3 shows LUBM query 12. Recall that a triple pattern cannot take part in two joins on two variables in a job. So the query can not be answered by a single job because of the third triple pattern (?X ub:worksFor ?Y). We can see that there are multiple ways to answer the query. One plan can be to join triple patterns 1 (?X rdf:type ub:Chair) and 3 (?X ub:worksFor ?Y) on variable X in first job, join triple patterns 2 (?Y rdf:type ub:Department) and 4 (?Y ub:subOrganizationOf <http://www.U0.edu>) on variable Y in second job and join the outputs of these two Jobs on variable Y in the third and final job. Another plan can be to join triple patterns 2, 3 and 4 on variable Y in first Job and join triples pattern 1 and the output of first job on variable X in the second and final job. Intuitively, we can say that the more the number of jobs, the slower the answering time is. This is because a job has a setup time and it reads and writes data from and to disk several times. So the first plan is slower than the second one. Hence we can see that there is a need to determine the best query plan before answering a SPARQL query.

2) Plan Generation by Graph Coloring: There are a couple of approaches to generate a query plan for a SPARQL query: greedy approach [11], exhaustive approach etc. The greedy approach is a simple one and generates a plan very quickly. At each step, it selects a variable on which maximum number of joins can be done. But it is not guaranteed that this approach will always generate the best plan. In this section, we present an algorithm which exhaustively searches for the best query plan.

To find the best query plan, we generate all possible plans. In each plan, there are one or more jobs. Each job does one or more joins. It even can do all joins, in which case we get a one job plan. To generate all such combinations, for each join we deal with two possibilities: either to consider it in a job or not consider it. For a job, we would like to select a subset of NCJs. We dynamically determine the number of jobs needed to complete the plan. Note that there may exist dependency between jobs. Therefore we need to execute them in sequence. Once the plan is generated, we determine the cost using the cost model discussed in the extension of [11].

This can be represented by a graph coloring problem. Graph coloring is the problem of assigning some colors to the vertices of a graph according to a specified set of constraints. An example of such a constraint is no two adjacent vertices can have same color. Similarly, in our case, conflicting join operations (CJ) can be represented by having an edge between them which prohibits both of them having the color $WHITE$. Thus, the only constraint we have in our problem is two adjacent nodes in a join graph can not have the color $WHITE$ together. Initially, we assign a default color, $GREY$, to all joins. Then, if we consider a join in a job, we color it $WHITE$; otherwise, we color it $BLACK$. Recall that vertices having NCJ edges can be colored as $WHITE$ and participate in a single job. In this way, it becomes a variation of a graph 2-coloring problem. The difference between our problem and graph 2-coloring problem is that we allow two adjacent nodes in join graph to have the color $BLACK$. We explain how our backtracking algorithm generates all possible combinations in the extension of [11].

V. Results

In this section we first present performance comparison between our framework and Jena In-Memory and SDB models and BigOWLIM. Next, we present our experiment with varying number of reducers to understand its impact on query runtime.

A. Comparison with Other Frameworks

We compared our implementation with the Jena In-Memory model and the SDB models and BigOWLIM for Queries 2, 9 and 12 from the LUBM dataset. The Jena models and BigOWLIM were tested on a machine having 2.80 GHz **quad** core processor, 8 GB main memory and 1 TB disk space. It is to be noted that BigOWLIM needed **7 GB** of Java heap space to successfully load the billion triples dataset. Our approach was tested on cluster of 10 nodes with POS schema. Each node had the same configuration: Pentium IV 2.80 GHz processor, 4 GB main memory and 640 GB disk space. Figures 3, 4 and 5 show the performance comparison for the queries respectively. In each these figures, the X axis represents number of triples in millions and the Y axis represents the time in seconds. We ran BigOWLIM only for the biggest three datasets as we are interested in its performance with large datasets only. For each set, on the X axis, there are four columns which show the results of Jena In-Memory model, Jena SDB model, our Hadoop implementation and BigOWLIM respectively. A cross represents either that the query could not complete or that it ran out of memory. In most of the cases, our approach was the fastest. For Query 2, Jena In-Memory Model and Jena SDB model were faster than our approach, giving results in 3.9 and 0.4 seconds respectively. However, as the size of the dataset grew, Jena In-Memory model ran out of memory space. Our implementation was much faster than Jena SDB model for large datasets. For example, in Figure 3 for 110 million triples, our approach took 143.5 seconds as compared to about 5000 seconds for Jena-SDB model. In Figures 4 and 5, we can see that Jena SDB model could not finish answering Queries 9 and 12. Jena In-Memory Model worked well for small datasets but became slower than our implementation as the dataset size grew and eventually ran out of memory. For Query

Table III
QUERY RUNTIMES FOR LUBM AND SP2B DATASET

Triples	LUBM-DATASET						SP2B-DATASET		
Billions	Q-1	Q-2	Q-4	Q-9	Q-12	Q-13	Q-1	Q-2	Q-3a
0.11	66.3	143.5	164.4	222.4	67.4	79.5	82.7	590.9	85.2
0.22	87.5	213.4	287.4	372.3	76	104.7	124.4	851	133.4
0.33	111.6	289.6	428.4	561.4	93.3	135.3	156.6	1229.7	159.5
0.44	129.7	360.4	586.1	675	115.1	160.8	198.2	1697.1	188.8
0.55	153.7	421.2	729.1	824.1	135	182	229.2	2119.4	225.1
0.66	176.9	498	871.4	1035.9	142.5	208.9	276.2	2643.5	260.7
0.77	195	570.6	1027.3	1192.2	149.8	236.8	314.3	3084.1	293.9
0.88	214.1	643.2	1135.5	1353	173.5	267.4	351.2	3528.3	331.7
0.99	229.2	718.3	1367.6	1503.2	200.5	294.5	384.7	4031.7	366.7
1.1	248.3	801.9	1430.2	1663.4	204.4	325.4	436.8	4578.7	402.6

2 (Figure 3), BigOWLIM was slower than us for the 110 and 550 million datasets. For 550 million dataset it took **22693.4** seconds, which is abruptly high compared to its other timings. For the billion triple dataset, it was faster than us. It should be noted that our framework does not have any indexing or triple cache whereas BigOWLIM exploits indexing which it loads to main memory when it starts. It may also pre-fetch triples to main memory. For Query 9 (Figure 4), our implementation is faster than BigOWLIM in all experiments. For Query 12 (Figure 5), BigOWLIM is faster. It is because their indexing helps them to answer query very fast when there is a triple pattern with bound object and the last triple pattern (?Y ub:subOrganizationOf <http://www.University0.edu>) of Query 12 is such a triple pattern. We should note that the loading time for BigOWLIM is significantly higher than ours. For example, for 0.11 billion triples dataset, it takes BigOWLIM 2.89 hours to load when it is first started. On the other hand, our framework has no loading time because after we insert the data for the first time we don't do any loading each time our framework or the cluster is restarted. In our experiments, the Hadoop cluster takes less than 1 minute to start up and as soon as the cluster is up, our framework is ready. If we consider the loading time as a part of query answering time, our framework is always faster than BigOWLIM. Excluding loading time, we are faster when there is no bound object in a SPARQL query.

Table III shows query time to execute the plan generated using *GenerateBestPlan* algorithm on different number of Triples. The first column represents the number of triples in the range between 110 million to 1100 million. Columns 2 to 7 of Table III represent the six selected queries from the LUBM dataset whereas the last three columns are the queries from SP2B dataset [20]. Query answering time is in seconds. The number of triples are rounded down to millions. As expected, as the number of triples increased, the time to answer a query also increased. For example, Query 1 for 100 million triples took 66.3 seconds whereas for 1100 million triples it took 248.3 seconds. Query 1 is simple and requires only one join, thus it took the least amount of time among all the queries. Query 2 is one of the two queries having the greatest number of triple patterns. We can observe that even though it has three times more triple patterns, it does not take thrice the time of Query 1 answering time because of our storage schema. Query 4 has one less triple pattern than Query 2, but it requires inferencing to bind 1 triple pattern. As we determine inferred relations on the fly, queries requiring inference takes longer times in our framework. Query 9 and 12 also require inference and Query 13 has an inverse property in one of its triple patterns.

As the size of the dataset grows, the increase in time to answer a query does not grow proportionately. The increase in time is always less. For example, there are ten times as many triples in the dataset of 10000 universities than 1000 universities, but for Query 1 the time only increases by **3.76** times and for query 9 by **7.49** times. The latter is the highest increase in time, yet it is still less than the increase in the size of the datasets. Due to space limitations, we do not report query runtimes with PS schema here. We observed that PS schema is much slower than POS schema.

B. Experiment with Number of Reducers

In a Hadoop job, the user can specify the number of reducers to use. We decided to find the impact of the number of reducers in answering a SPARQL query. For the six queries discussed in Section V-A, we varied the number of reducers from 1 to 10. We can see the run times in Figure 6. The horizontal axis shows the number of reducers and the vertical one shows time in milliseconds. We can see a clear trend in the run times. As we increase the number of reducers, queries are answered faster. This can be explained by the reducer loads. The less the number of reducers, the more keys are sent to a reducer. As after the map phase the query run time is determined by the slowest running reducer, the more a reducer gets load the slower the query is. That is why queries with only 1 reducer are slowest and with 10 reducers are the fastest ones. In the figure, we see that queries 2, 4 and 9 become much faster as the number of reducers are increased than queries 1, 12 and 13. It is because the size of the output of the map phase of the former

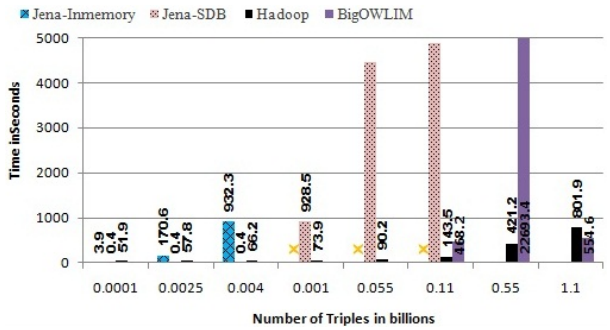

Figure 3. Response time of LUBM Query 2

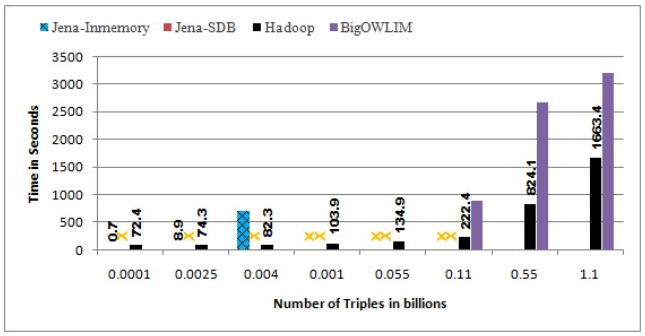

Figure 4. Response time of LUBM Query 9

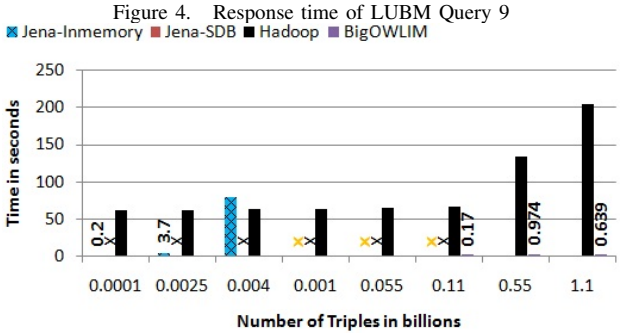

Figure 5. Response time of LUBM Query 12

queries are significantly large whereas for the latter group the size of the output of that phase is so small that less number of reducers are adequate to handle it efficiently.

C. Experiment with PigLatin

We tested our framework on cluster of 10 nodes with POS schema. Each node had the same configuration: Pentium IV 2.80 GHz processor, 4 GB main memory and 640 GB disk space. Figure 7 shows the times taken by our framework to answer six of the benchmark queries. Due to space constraints, we do not report the rest of the queries. We can see that for queries 1, 2, 12 and 13 the increment in runtime when the data set size increases is very small. Query 4 needs only one PigLatin JOIN statement but because of inference it has a very large input. Query 9 needs two PigLatin JOIN statements and has a large input because of inference.

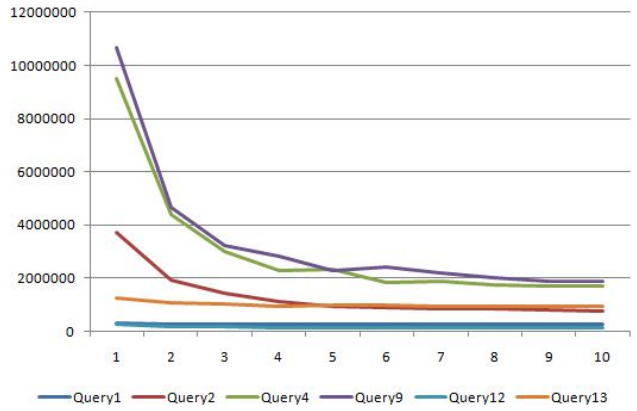

Figure 6. Query Time vs. Number of Reducers

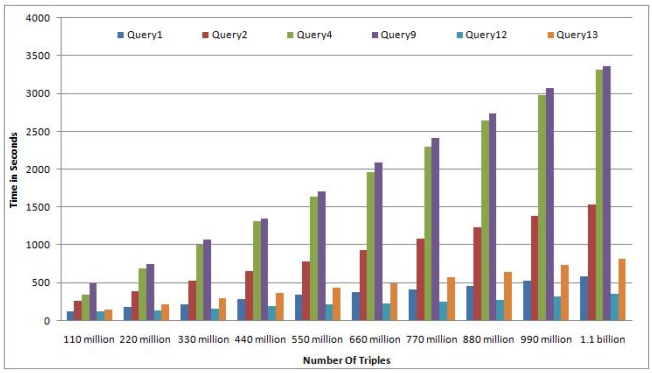

Figure 7. Query Runtimes

In Table IV we have shown the comparison of runtimes to answer LUBM queries 1 and 2 on datasets of different sizes between our approach and Yahoo! Research's Pig module plugged in Sesame framework [14]. We can see that in all cases our framework performed much better than theirs. For query 1 on data set of 50 universities, we are 64% faster. For the same query on data set of 100 universities, we are 75% faster, which indicates that as the size of the data set grows, our framework performs even more better. For query 2 on same data set we have the highest gain.

We note that Figure 1 of [14], for 10 node cluster query execution time for LUBM query 1 on 1000 universities dataset is not shown as it is much higher than 600 seconds which is the maximum runtime shown in the graph. In our experiments, we took 248.3 seconds to answer the same

Query	Data sets	Our Approach	Yahoo! Research	% Gain
Query 1	50	32	90	64
Query 1	100	41	170	75
Query 2	100	104	550	81

Table IV
QUERY RUNTIMES COMPARISON

query which is substantially less than even 600 seconds.

VI. Conclusions And Future Works

We have presented a framework capable of handling enormous amount of RDF data. Since our framework is based on Hadoop, which is a distributed and highly fault tolerant system, it inherits those two properties automatically. The framework is highly scalable. To increase capacity of our system all that needs to be done is to add new nodes to the Hadoop cluster. We have proposed a schema to store RDF data in plain text files, an algorithm to determine the best processing plan to answer a SPARQL query and a cost model to be used by the algorithm. Our experiments demonstrate that our system is highly scalable. If we add data, the delay introduced to answer a query does not increase as much as the increment in the data size.

In the future, we would extend the work in few directions. First, we will build a cloud service based on the framework. Second, we will investigate the case when POS step does not produce a balanced partitioning because of a skewed distribution of the data. Finally, we will experiment with non-homogenous cluster and address the challenges we face.

Acknowledgment

This research was funded in part by IARPA and Raytheon Company.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden and Kate Hollenbach, *SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management*, VLDB Journal, 18(2), April, 2009.

[2] Daniel J. Abadi, Adam Marcus, Samuel R. Madden and Kate Hollenbach, *Scalable semantic web data management using vertical partitioning*, VLDB 2007.

[3] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz and Alexander Rasin, *HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads*, VLDB 2009.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, Seventh Symposium on Operating System Design and Implementation, November, 2006.

[5] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, A. Y. Ng and K. Olukotun, *Map-reduce for machine learning on multicore*, NIPS, 2007.

[6] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: simplified data processing on large clusters*, OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, 2004.

[7] Li Ding, Tim Finin, Yun Peng, Paulo P. da Silva and Deborah L. Mcguinness, *Tracking RDF Graph Provenance using RDF Molecules*, Proc. of the 4th International Semantic Web Conference (Poster), 2005.

[8] Y. Guo, Z. Pan and J. Heflin, *LUBM: A Benchmark for OWL Knowledge Base Systems*, Journal of Web Semantics, 2005.

[9] Yuanbo Guo and Jeff Heflin, *A Scalable Approach for Partitioning OWL Knowledge Bases*, 2nd Int. Workshop on Scalable Semantic Web Knowledge Base Systems, 2006.

[10] Yuanbo Guo, Zhengxiang Pan and Jeff Heflin, *An evaluation of knowledge base systems for large OWL datasets*, International Semantic Web Conference, 2004.

[11] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan and Bhavani Thuraisingham, *Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce*, CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing, 2009.

[12] Andrew W. Mcnabb, Christopher K. Monson and Kevin D. Seppi, *MRPSO: MapReduce particle swarm optimization*, GECCO 2007.

[13] M. Michael, J.E. Moreira, D. Shiloach and R. W. Wisniewski, *Scale-up x Scale-out: A Case Study using Nutch/Lucene*, IPDPS 2007.

[14] P. Mika and G. Tummarello, *Web Semantics in the Clouds*, IEEE Intelligent Systems, Volume 23, 2008.

[15] Christopher Moretti, Karsten Steinhaeuser, Douglas Thain and Nitesh V. Chawla, *Scaling Up Classifiers to Cloud Computers*, ICDM, 2008.

[16] A. Newman, J. Hunter, Y. F. Li, C. Bouton and M. Davis, *A Scale-Out RDF Molecule Store for Distributed Processing of Biomedical Data*, Semantic Web for Health Care and Life Sciences Workshop, WWW 2008.

[17] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar and Andrew Tomkins, *Pig Latin: A Not-So-Foreign Language for Data Processing*, SIGMOD, 2008.

[18] Patrick Pantel, *Data Catalysis: Facilitating Large-Scale Natural Language Data Processing*, ISUC 2007.

[19] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder and John Sumner, *An Evaluation of Triple-Store Technologies for Large Data Stores*, On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops.

[20] Michael Schmidt, Thomas Hornung, Georg Lausen and Christoph Pinkel, *SP2Bench: A SPARQL Performance Benchmark*, ICDE 2009.

[21] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer and Dave Reynolds, , *SPARQL basic graph pattern optimization using selectivity estimation*, WWW 2008.

[22] Jacopo Urbani, Spyros Kotoulas, Eyal Oren and Frank van Harmelen, *Scalable Distributed Reasoning Using MapReduce*, International Semantic Web Conference, 2009.

[23] Jesse Weaver and James A. Hendler, *Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples*, ISWC 2009.