Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing

Mohammad Farhan Husain, James McGlothlin, Mohammad Mehedy Masud, Latifur R. Khan, and Bhavani Thuraisingham, *Fellow*, *IEEE*

Abstract—Semantic web is an emerging area to augment human reasoning. Various technologies are being developed in this arena which have been standardized by the World Wide Web Consortium (W3C). One such standard is the Resource Description Framework (RDF). Semantic web technologies can be utilized to build efficient and scalable systems for Cloud Computing. With the explosion of semantic web technologies, large RDF graphs are common place. This poses significant challenges for the storage and retrieval of RDF graphs. Current frameworks do not scale for large RDF graphs and as a result do not address these challenges. In this paper, we describe a framework that we built using Hadoop to store and retrieve large numbers of RDF triples by exploiting the cloud computing paradigm. We describe a scheme to store RDF data in Hadoop Distributed File System. More than one Hadoop job (the smallest unit of execution in Hadoop) may be needed to answer a query because a single triple pattern in a query cannot simultaneously take part in more than one join in a single Hadoop job. To determine the jobs, we present an algorithm to generate query plan, whose worst case cost is bounded, based on a greedy approach to answer a SPARQL Protocol and RDF Query Language (SPARQL) query. 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 handle large amounts of RDF data, unlike traditional approaches.

Index Terms—Hadoop, RDF, SPARQL, MapReduce.

1 Introduction

Cloud computing is an emerging paradigm in the IT and data processing communities. Enterprises utilize cloud computing service to outsource data maintenance, which can result in significant financial benefits. Businesses store and access data at remote locations in the "cloud." As the popularity of cloud computing grows, the service providers face ever increasing challenges. They have to maintain huge quantities of heterogenous data while providing efficient information retrieval. Thus, the key emphasis for cloud computing solutions is scalability and query efficiency.

Semantic web technologies are being developed to present data in standardized way such that such data can be retrieved and understood by both human and machine. Historically, webpages 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. Cloud Computing systems can utilize the power of these Semantic web technologies to provide the user with capability to efficiently store and retrieve data for data intensive applications.

Semantic web technologies could be especially useful for maintaining data in the cloud. Semantic web technologies provide the ability to specify and query heterogenous data in a standardized manner. Moreover, via Web Ontology Language (OWL) ontologies, different schemas, classes, data types, and relationships can be specified without sacrificing the standard RDF/SPARQL interface. Conversely, cloud computing solutions could be of great benefit to the semantic web community. Semantic web data sets are growing exponentially. More than any other arena, in the web domain, scalability is paramount. Yet, high speed response time is also vital in the web community. We believe that the cloud computing paradigm offers a solution that can achieve both of these goals.

Existing commercial tools and technologies do not scale well in Cloud Computing settings. Researchers have started to focus on these problems recently. They are proposing systems built from the scratch. In [39], researchers propose an indexing scheme for a new distributed database[3] which can be used as a Cloud system. When it comes to semantic web data such as RDF, we are faced with similar challenges. With storage becoming cheaper and the need to store and retrieve large amounts of data, developing systems to

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

2. http://www.w3.org/TR/rdf-sparql-query.

3. http://www.comp.nus.edu.sg/~epic/.

- *M.F. Husain is with Amazon.com, 2201 Westlake Avenue, Seattle, WA 98121. E-mail: mfh062000@utdallas.edu.*
- *J. McGlothlin, M.M. Masud, L.R. Khan, and B. Thuraisingham are with the University of Texas at Dallas, 800 W. Campbell Road, Richardson, TX 75080. E-mail: {jpm083000, mehedy, lkhan, bhavani.thuraisingham}@utdallas.edu.*

Manuscript received 2 Apr. 2010; revised 28 Oct. 2010; accepted 23 Feb. 2011; published online 27 Apr. 2011.
Recommended for acceptance by D. Lomet.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-2010-04-0196.
Digital Object Identifier no. 10.1109/TKDE.2011.103.

handle billions of RDF triples requiring tera bytes of disk space is no longer a distant prospect. Researchers are already working on billions of triples [30], [33]. Competitions are being organized to encourage researchers to build efficient repositories.[4] At present, there are just a few frameworks (e.g., RDF-3X [29], Jena [7], Sesame,[5] BigOWLIM [22]) for Semantic web technologies, and these frameworks have limitations for large RDF graphs. Therefore, storing a large number of RDF triples and efficiently querying them is a challenging and important problem.

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. 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. SPARQL queries can be answered by converting them to SQL first [9], [10], [12]. Optimal relational schemas are being probed for this purpose [3]. The main disadvantage with such systems is that they are optimized for relational data. They may not perform well for RDF data, especially because RDF data are sets of triples[6] (an ordered tuple of three components called subject, predicate, and object, respectively) which form large directed graphs. In an SPARQL query, any number of triple patterns (TPs)[7] can join on a single variable[8] which makes a relational database query plan complex. Performance and scalability will remain a challenging issue due to the fact that these systems are optimized for relational data schemata and transactional database usage.

Yet another approach is to build a distributed system for RDF from scratch. Here, there will be an opportunity to design and optimize a system with specific application to RDF data. In this approach, the researchers would be reinventing the wheel.

Instead of starting with a blank slate, we propose to build a solution with a generic distributed storage system which utilizes a Cloud Computing platform. We then propose to tailor the system and schema specifically to meet the needs of semantic web data. Finally, we propose to build a semantic web repository using such a storage facility.

Hadoop[9] is a distributed file system where files can be saved with replication. It is 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 [13] programming model, a functional programming model which is suitable for the parallel processing of large amounts of data. Through partitioning data into a number of independent chunks, MapReduce processes run against these chunks, making parallelization simpler. Moreover, the MapReduce programming model facilitates and simplifies the task of joining multiple triple patterns.

In this paper, we will describe a schema to store RDF data in Hadoop, and we will detail a solution to process queries against these data. In the preprocessing stage, we process RDF data and populate files in the distributed file system. This process includes partitioning and organizing the data files and executing dictionary encoding.

We will then detail a query engine for information retrieval. We will specify exactly how SPARQL queries will be satisfied using MapReduce programming. Specifically, we must determine the Hadoop "jobs" that will be executed to solve the query. We will present a greedy algorithm that produces a query plan with the minimal number of Hadoop jobs. This is an approximation algorithm using heuristics, but we will prove that the worst case has a reasonable upper bound.

Finally, we will utilize two standard benchmark data sets to run experiments. We will present results for data set ranging from 0.1 to over 6.6 billion triples. We will show that our solution is exceptionally scalable. We will show that our solution outperforms leading state-of-the-art semantic web repositories, using standard benchmark queries on very large data sets.

Our contributions are as follows:

1. We design a storage scheme to store RDF data in Hadoop distributed file system (HDFS[10]).
2. We propose an algorithm that is guaranteed to provide a query plan whose cost is bounded by the log of the total number of variables in the given SPARQL query. It uses summary statistics for estimating join selectivity to break ties.
3. We build a framework which is highly scalable and fault tolerant and supports data intensive query processing.
4. We demonstrate that our approach performs better than Jena for all queries and BigOWLIM and RDF-3X for complex queries having large result sets.

The remainder of this paper is organized as follows: in Section 2, we investigate related work. In Section 3, we discuss our system architecture. In Section 4, we discuss how we answer an SPARQL query. In Section 5, we present the results of our experiments. Finally, in Section 6, we draw some conclusions and discuss areas we have identified for improvement in the future.

2 Related Work

MapReduce, though a programming paradigm, is rapidly being adopted by researchers. This technology is becoming increasingly popular in the community which handles large amounts of data. It is the most promising technology to solve the performance issues researchers are facing in Cloud Computing. In [1], Abadi discusses how MapReduce can satisfy most of the requirements to build an ideal Cloud DBMS. Researchers and enterprises are using MapReduce technology for web indexing, searches, and data mining. In

4. http://challenge.semanticweb.org.
5. http://www.openrdf.org.
6. http://www.w3.org/TR/rdf-concepts/#dfn-rdf-triple.
7. http://www.w3.org/TR/rdf-sparql-query/#defn_TriplePattern.
8. http://www.w3.org/TR/rdf-sparql-query/#defn_QueryVariable.
9. http://hadoop.apache.org.

10. http://hadoop.apache.org/core/docs/r0.18.3/hdfs_design.html.

this section, we will first investigate research related to MapReduce. Next, we will discuss works related to the semantic web.

Google uses MapReduce for web indexing, data storage, and social networking [8]. Yahoo! uses MapReduce extensively in its data analysis tasks [31]. IBM has successfully experimented with a scale-up scale-out search framework using MapReduce technology [27]. In a recent work [35], they have reported how they integrated Hadoop and System R. Teradata did a similar work by integrating Hadoop with a parallel DBMS [42].

Researchers have used MapReduce to scale up classifiers for mining petabytes of data [28]. 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 [11], 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 [25]. In [4], 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. Our focus is on developing a scalable solution for storing RDF data and retrieving them by SPARQL queries.

In the semantic web arena, there has not been much work done with MapReduce technology. We have found two related projects: BioMANTA[11] project and Scalable, High-Performance, Robust and Distributed (SHARD).[12] BioMANTA proposes extensions to RDF Molecules [14] and implements a MapReduce-based Molecule store [30]. They use MapReduce to answer the queries. They have queried a maximum of four million triples. Our work differs in the following ways: first, we have queried one 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 a query processing plan whose cost is bounded by the log of the total number of variables in the given SPARQL query. By using this, 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 an SPARQL query.

SHARD is an RDF triple store using the Hadoop Cloudera distribution. This project shows initial results demonstrating Hadoop's ability to improve scalability for RDF data sets. However, SHARD stores its data only in a triple store schema. It currently does no query planning or reordering, and its query processor will not minimize the number of Hadoop jobs.

There has been significant research into semantic web repositories, with particular emphasis on query efficiency and scalability. In fact, there are too many such repositories to fairly evaluate and discuss each. Therefore, we will pay attention to semantic web repositories which are open source or available for download, and which have received favorable recognition in the semantic web and database communities.

In [2] and [3], researchers reported a vertically partitioned DBMS for storage and retrieval of RDF data. Their solution is a schema with a two-column table for each predicate. Their schema is then implemented on top of a column-store relational database such as CStore [37] or MonetDB [6]. They observed performance improvement with their scheme over traditional relational database schemes. We have leveraged this technology in our predicate-based partitioning within the MapReduce framework. However, in the vertical partitioning research, only small databases (<100 million) were used. Several papers [16], [23], [41] have shown that vertical partitioning's performance is drastically reduced as the data set size is increased.

Jena [7] is a semantic web framework for Jena. True to its framework design, it allows integration of multiple solutions for persistence. It also supports inference through the development of *reasoners*. However, Jena is limited to a triple store schema. In other words, all data are stored in a single three-column table. Jena has very poor query performance for large data sets. Furthermore, any change to the data set requires complete recalculation of the inferred triples.

BigOWLIM [22] is among the fastest and most scalable semantic web frameworks available. However, it is not as scalable as our framework and requires very high end and costly machines. It requires expensive hardware (a lot of main memory) to load large data sets and it has a long loading time. As our experiments show (Section 5.4), 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.

RDF-3X [29] is considered the fastest existing semantic web repository. In other words, it has the fastest query times. RDF-3X uses histograms, summary statistics, and query optimization to enable high performance semantic web queries. As a result, RDF-3X is generally able to outperform any other solution for queries with bound objects and aggregate queries. However, RDF-3X's performance degrades exponentially for unbound queries, and queries with even simple joins if the selectivity factor is low. This becomes increasingly relevant for inference queries, which generally require unions of subqueries with unbound objects. Our experiments show that RDF-3X is not only slower for such queries, it often aborts and cannot complete the query. For example, consider the simple query "Select all students." This query in LUBM requires us to select all graduate students, select all undergraduate students, and union the results together. However, there are a very large number of results in this union. While both subqueries complete easily, the union will abort in RDF-3X for LUBM (30,000) with 3.3 billion triples.

RDF Knowledge Base (RDFKB) [24] is a semantic web repository using a relational database schema built upon bit vectors. RDFKB achieves better query performance than RDF-3X or vertical partitioning. However, RDFKB aims to provide knowledge base functions such as inference forward chaining, uncertainty reasoning, and ontology alignment. RDFKB prioritizes these goals ahead of scalability. RDFKB is not able to load LUBM (30,000) with three billion triples, so it cannot compete with our solution for scalability.

Hexastore [41] and BitMat [5] are main memory data structures optimized for RDF indexing. These solutions may achieve exceptional performance on hot runs, but they

11. http://www.itee.uq.edu.au/eresearch/projects/biomanta.
12. http://www.cloudera.com/blog/2010/03/how-raytheon-researchers-are-using-hadoop-to-build-a-scalable-distributed-triple-store.

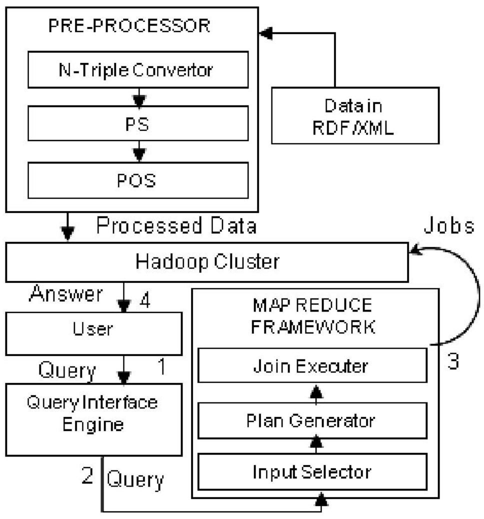


Fig. 1. The system architecture.

are not optimized for cold runs from persistent storage. Furthermore, their scalability is directly associated with the quantity of main memory RAM available. These products are not available for testing and evaluation.

In our previous works [20], [21], we proposed a greedy and an exhaustive search algorithm to generate a query processing plan. However, the exhaustive search algorithm was expensive and the greedy one was not bounded and its theoretical complexity was not defined. In this paper, we present a new greedy algorithm with an upper bound. Also, we did observe scenarios in which our old greedy algorithm failed to generate the optimal plan. The new algorithm is able to obtain the optimal plan in each of these cases. Furthermore, in our prior research, we were limited to text files with minimal partitioning and indexing. We now utilize dictionary encoding to increase performance. We have also now done comparison evaluation with more alternative repositories.

3 Proposed Architecture

Our architecture consists of two components. The upper part of Fig. 1 depicts the data preprocessing component and the lower part shows the query answering one.

We have three subcomponents for data generation and preprocessing. We convert RDF/XML[13] to N-Triples[14] serialization format using our N-Triples Converter component. The Predicate Split (PS) component takes the N-Triples data and splits it into predicate files. The predicate files are then fed into the Predicate Object Split (POS) component which splits the predicate files into smaller files based on the type of objects. These steps are described in Sections 3.2, 3.3, and 3.4.

Our MapReduce framework has three subcomponents in it. It takes the SPARQL query from the user and passes it to the Input Selector (see Section 4.1) and Plan Generator. This component selects the input files, by using our algorithm described in Section 4.3, 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.

3.1 Data Generation and Storage

For our experiments, we use the LUBM [18] data set. It is a benchmark data set designed to enable researchers to evaluate a semantic web repository's performance [19]. 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 would 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 into our intended format are described in following sections.

3.2 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 the 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.

3.3 Predicate Split

In the first step, we divide the data according to the predicates. This division immediately enables us to cut down the search space for any SPARQL query which does not have a variable[15] 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[16] 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 4.1. In real-world RDF data sets, the number of distinct predicates is in general not a large number [36]. However, there are data sets having many predicates. Our system performance does not vary in such a case because we just select files related to the predicates specified in a SPARQL query.

3.4 Predicate Object Split

3.4.1 Split Using Explicit Type Information of Object

In the next step, we work with the explicit type information in the *rdf_type* file. The predicate *rdf:type* is used in RDF to denote that a resource is an instance of a class. The *rdf_type* 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 $c_1, c_2, \ldots, c_n$, then we will create files for each of these leaves and the file names will be like *type_*c_1, *type_*$c_2, \ldots,$ *type_*c_n. Please note that the object values $c_1, c_2, \ldots, c_n$ are no

13. http://www.w3.org/TR/rdf-syntax-grammar.
14. http://www.w3.org/2001/sw/RDFCore/ntriples.

15. http://www.w3.org/TR/rdf-sparql-query/#sparqlQueryVariables.
16. http://www.w3.org/TR/rdf-sparql-query/#sparqlTriplePatterns.

TABLE 1
Data Size at Various Steps for 1,000 Universities

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

longer needed to be stored within 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*.

3.4.2 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 *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 appear in one line in the file p_c_i. To do this split, we need to join a predicate file with the *type_** files to retrieve the type information.

In Table 1, 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.

Table 1 shows the number of files and size gain we get at each step for data from 1,000 universities. LUBM generator generates 20,020 small files, with a total size of 24 GB. After splitting the data according to predicates, the size drastically reduces to only 7.1 GB (a 70.42 percent gain). This happens because of the absence of predicate columns and also the prefix substitution. At this step, we have only 17 files as there are 17 unique predicates in the LUBM data set. In the final step, space is reduced another 7.04 percent, as the split *rdf-type* files no longer have the object column. The number of files increases to 41 as predicate files are split using the type of the objects.

3.5 Example Data

In Table 2, we have shown sample data for three predicates. The leftmost 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 $advisor$, $takesCourse$, 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.

3.6 Binary Format

Up to this point, we have shown our files in text format. Text format is the natively supported format by Hadoop. However, for increased efficiency, storing data in binary format is an option. We do dictionary encoding to encode the strings with a long value (64-bit). In this way, we are able to store up to 2^{64} unique strings. We dictionary encode the data using Hadoop jobs. We build a prefix tree in each reducer and generate a unique id for a string by using the reducer id, which is unique across the job. We generate the dictionary in one job and then run three jobs to replace the subject, predicate, and object of a triple with their corresponding id as text. In the final job, we convert the triples consisting of ids in text to binary data.

TABLE 2
Sample Data for LUBM Query 9

type_Student	ub:advisor_FullProfessor	
GS1	GS2	A2
GS2	GS1	A1
GS3	GS3	A3

ub:takesCourse_Course		ub:teacherOf_Course	
GS1	C2	A1	C1
GS3	C1	A2	C2
GS2	C3	A3	C3

4 MapReduce Framework

In this section, we discuss how we answer SPARQL queries in our MapReduce framework component. Section 4.1 discusses our algorithm to select input files for answering the query. Section 4.2 talks about cost estimation needed to generate a plan to answer an SPARQL query. It introduces few terms which we use in the following discussions. Section 4.2.1 discusses the ideal model we should follow to estimate the cost of a plan. Section 4.2.2 introduces the heuristics-based model we use in practice. Section 4.3 presents our heuristics-based greedy algorithm to generate a query plan which uses the cost model introduced in Section 4.2.2. We face tie situations in order to generate a plan in some cases and Section 4.4 talks about how we handle these special cases. Section 4.5 shows how we implement a join in a Hadoop MapReduce job by working through an example query.

4.1 Input Files Selection

Before determining the jobs, we select the files that need to be inputted to the jobs. We have some query rewriting capability which we apply at this step of query processing. We take the query submitted by the user and iterate over the triple patterns. We may encounter the following cases:

1. In a triple pattern, if the predicate is variable, we select all the files as input to the jobs and terminate the iteration.
2. If the predicate is *rdf:type* and the object is concrete, we select the *type* file having that particular type. For example, for LUBM query 9 (Listing 1), we could select file *type_Student* as part of the input set. However, this brings up an interesting scenario. In our data set, there is actually no file named *type_Student* because *Student* class is not a leaf in the ontology tree. In this case, we consult the LUBM ontology,[17] part of which is shown in Fig. 2, to

17. http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl.

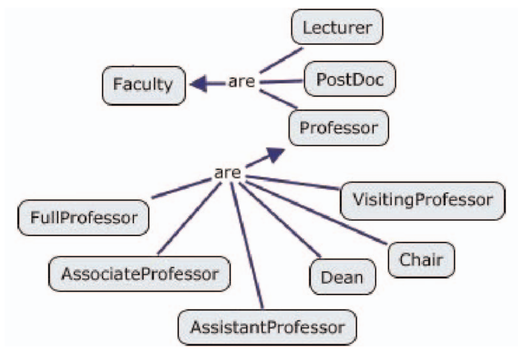


Fig. 2. Partial LUBM ontology (**are** denotes subClassOf relationship).

determine the correct set of input files. We add the files *type_GraduateStudent*, *type_UndergraduateStudent*, and *type_ResearchAssistant* as *GraduateStudent*, *UndergraduateStudent*, and *ResearchAssistant* are the leaves of the subtree rooted at node *Student*.

3. If the predicate is *rdf:type* and the object is variable, then if the type of the variable is defined by another triple pattern, we select the *type* file having that particular type. Otherwise, we select all *type* files.
4. If the predicate is not *rdf:type* and the object is variable, then we need to determine if the type of the object is specified by another triple pattern in the query. In this case, we can rewrite the query eliminate some joins. For example, in LUBM Query 9 (Listing 1), the type of Y is specified as *Faculty* and Z as *Course* and these variables are used as objects in last three triple patterns. If we choose files *advisor_Lecturer*, *advisor_PostDoc*, *advisor_FullProfessor*, *advisor_AssociateProfessor*, *advisor_AssistantProfessor*, and *advisor_ VisitingProfessor* as part of the input set, then the triple pattern in line 2 becomes unnecessary. Similarly, triple pattern in line 3 becomes unnecessary if files *takesCourse_Course* and *takesCourse_GraduateCourse* are chosen. Hence, we get the rewritten query shown in Listing 2. However, if the type of the object is not specified, then we select all files for that predicate.
5. If the predicate is not *rdf:type* and the object is concrete, then we select all files for that predicate.

Listing 1. LUBM Query 9

```
SELECT ?X ?Y ?Z WHERE {
?X rdf:type ub:Student.
?Y rdf:type ub:Faculty.
?Z rdf:type ub:Course.
?X ub:advisor ?Y.
?Y ub:teacherOf ?Z.
?X ub:takesCourse ?Z}
```

Listing 2. Rewritten LUBM Query 9

```
SELECT ?X ?Y ?Z WHERE {
?X rdf:type ub:Student.
?X ub:advisor ?Y.
?Y ub:teacherOf ?Z.
?X ub:takesCourse ?Z}
```

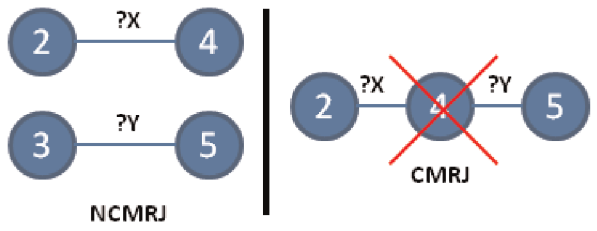


Fig. 3. NCMRJ and CMRJ example.

4.2 Cost Estimation for Query Processing

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

Definition 1 (Triple Pattern, TP). *A triple pattern is an ordered set of subject, predicate, and object which appears in an SPARQL query WHERE clause. The subject, predicate, and object can be either a variable (unbounded) or a concrete value (bounded).*

Definition 2 (Triple Pattern Join, TPJ). *A triple pattern join is a join between two TPs on a variable.*

Definition 3 (MapReduceJoin, MRJ). *A MapReduceJoin is a join between two or more triple patterns on a variable.*

Definition 4 (Job, JB). *A job JB is a Hadoop job where one or more MRJs are done. JB has a set of input files and a set of output files.*

Definition 5 (Conflicting MapReduceJoins, CMRJ). *Conflicting MapReduceJoins is a pair of MRJs on different variables sharing a triple pattern.*

Definition 6 (Nonconflicting MapReduceJoins, NCMRJ). *Nonconflicting MapReduceJoins is a pair of MRJs either not sharing any triple pattern or sharing a triple pattern and the MRJs are on same variable.*

An example will illustrate these terms better. In Listing 3, we show LUBM Query 12. Lines 2, 3, 4, and 5 each have a triple pattern. The join between TPs in lines 2 and 4 on variable $?X$ is an MRJ. If we do two MRJs, one between TPs in lines 2 and 4 on variable $?X$ and the other between TPs in lines 4 and 5 on variable $?Y$, there will be a CMRJ as **TP in line 4 ($?X$ ub:worksFor $?Y$) takes part in two MRJs on two different variables $?X$ and $?Y$**. This is shown on the right in Fig. 3. This type of join is called CMRJ because in a Hadoop job, more than one variable of a TP cannot be a key at the same time and MRJs are performed on keys. An NCMRJ, shown on the left in Fig. 3, would be one MRJ between triple patterns in lines 2 and 4 on variable $?X$ and another MRJ between triple patterns in lines 3 and 5 on variable $?Y$. These two MRJs can make up a JB.

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}
```

4.2.1 Ideal Model

To answer an SPARQL query, we may need more than one job. Therefore, in an ideal scenario, the 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.

$$Cost = \left(\sum_{i=1}^{n-1} MI_i + MO_i + RI_i + RO_i \right) + MI_n + MO_n + RI_n \tag{1}$$

$$= \left(\sum_{i=1}^{n-1} Job_i \right) + MI_n + MO_n + RI_n \tag{2}$$

$$Job_i = MI_i + MO_i + RI_i + RO_i \quad (if \quad i < n). \tag{3}$$

Where,

$MI_i =$ Map Input phase for Job i.
$MO_i =$ Map Output phase for Job i.
$RI_i =$ Reduce Input phase for Job i.
$RO_i =$ 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 only the map phase of the final job. We do not consider the cost of the reduce output of the final job because it would be same for any query plan as this output is the final result which is fixed for a query and a given data set. 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 can estimate the cost for the MI phase to be equal to the total number of triples in each of the selected files.

Map Output (MO) phase. 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, a bound component selectivity estimation can be applied. 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 (RI) phase. 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 (RO) phase. The RO phase deals with performing the joins. Therefore, it is in this phase we can use the join triple pattern selectivity summary statistics to estimate the size of its output. Section 4.2.2 talks in detail about the join triple pattern selectivity summary statistics needed for our framework.

However, in practice, the above discussion is applicable for the first job only. For the subsequent jobs, we lack both the precise knowledge and estimate of the number of triple patterns selected after applying the join in the first job. Therefore, for these jobs, we can take the size of the RO phase of the first job as an upper bound on the different phases of the subsequent jobs.

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.

4.2.2 Heuristic Model

In this section, we show that the ideal model is not practical or cost effective. There are several issues that make the ideal model less attractive in practice. First, the ideal model considers simple abstract costs, namely, the number of triples read and written by the different phases ignoring the actual cost of copying, sorting, etc., these triples, and the overhead for running jobs in Hadoop. But accurately incorporating those costs in the model is a difficult task. Even making reasonably good estimation may be nontrivial. Second, to estimate intermediate join outputs, we need to maintain comprehensive summary statistics. In a MapReduce job in Hadoop, all the joins on a variable are joined together. For example, in the rewritten LUBM Query 9 (Listing 2), there are three joins on variable X. When a job is run to do the join on X, all the joins on X between triple patterns 1, 2, and 4 are done. If there were more than three joins on X, all will still be handled in one job. This shows that in order to gather summary statistics to estimate join selectivity, we face an exponential number of join cases. For example, between triple patterns having predicates p_1, p_2, and p_3, there may be 2^3 types of joins because in each triple pattern, a variable can occur either as a subject or an object. In the case of the rewritten Query 9, it is a subject-subject-subject join between 1, 2, and 4. There can be more types of join between these three, e.g., subject-object-subject, object-subject-object, etc. That means, between P predicates, there can be 2^P type of joins on a single variable (ignoring the possibility that a variable may appear both as a subject and object in a triple pattern). If there are P predicates in the data set, total number of cases for which we need to collect summary statistics can be calculated by the formula:

$$2^2 \times C_2^P + 2^3 \times C_3^P + \cdots + 2^P \times C_P^P.$$

In LUBM data set, there are 17 predicates. So, in total, there are 129,140,128 cases which is a large number. Gathering summary statistics for such a large number of cases would be very much time and space consuming. Hence, we took an alternate approach.

We observe that there is significant overhead for running a job in Hadoop. Therefore, if we minimize the number of jobs to answer a query, we get the fastest plan. The overhead is incurred by several disk I/O and network transfers that are integral part of any Hadoop job. When a job is submitted to Hadoop cluster, at least the following set of actions take place:

1. The Executable file is transferred from client machine to Hadoop JobTracker.[18]
2. The JobTracker decides which TaskTrackers[19] will execute the job.
3. The Executable file is distributed to the TaskTrackers over the network.
4. Map processes start by reading data from HDFS.
5. Map outputs are written to discs.
6. Map outputs are read from discs, shuffled (transferred over the network to TaskTrackers which would run Reduce processes), sorted, and written to discs.
7. Reduce processes start by reading the input from the discs.
8. Reduce outputs are written to discs.

These disk operations and network transfers are expensive operations even for a small amount of data. For example, in our experiments, we observed that the overhead incurred by one job is almost equivalent to **reading** a billion triples. The reason is that in every job, the output of the map process is always sorted before feeding the reduce processes. This sorting is unavoidable even if it is not needed by the user. Therefore, it would be less costly to process several hundred million more triples in n jobs, rather than processing several hundred million less triples in $n + 1$ jobs.

To further investigate, we did an experiment where we used the query shown in Listing 4. Here, the join selectivity between TPs 2 and 3 on $?Z$ is the highest. Hence, a query plan generation algorithm which uses selectivity factors to pick joins would select this join for the first job. As the other TPs 1 and 4 share variables with either TP 2 or 3, they cannot take part in any other join, moreover, they do not share any variables so the only possible join that can be executed in this job is the join between TPs 2 and 3 on $?X$. Once this join is done, the two joins left are between TP 1 and the join output of first job on variable $?X$ and between TP 4 and the join output of first job on variable $?Y$. We found that the selectivity of the first join is greater than the latter one. Hence, the second job will do this join and TP 4 will again not participate. In the third and last job, the join output of the second job will be joined with TP 4 on $?Y$. This is the plan generated using join selectivity estimation. But the minimum job plan is a 2-job plan where the first job joins TPs 1 and 2 on $?X$ and TPs 3 and 4 on $?Y$. The second and final job joins the two join outputs of the first job on $?Z$. The query runtimes we found are shown in Table 3 in seconds.

Listing 4. Experiment Query

?S1 ub:advisor ?X.
?X ub:headOf ?Z.
?Z ub:subOrganizationOf ?Y.
?S2 ub:mastersDegreeFrom ?Y

18. http://wiki.apache.org/hadoop/JobTracker.
19. http://wiki.apache.org/hadoop/TaskTracker.

TABLE 3
2-Job Plan versus 3-Job Plan

Dataset	2 Job Plan	3 Job Plan	Difference
LUBM_10000	4920	9180	4260
LUBM_20000	31020	36540	5520
LUBM_30000	80460	93947	13487

We can see that for each data set, the 2-job plan is faster than the 3-job plan even though the 3-job plan produced less intermediate data because of the join selectivity order. We can explain this by an observation we made in another small experiment. We generated files of sizes 5 and 10 MB containing random integers. We put the files in HDFS. For each file, we first read the file by a program and recorded the time needed to do it. While reading, our program reads from one of the three available replica of the file. Then, we ran a MapReduce job which rewrites the file with the numbers sorted. We utilized MapReduces sorting to have the sorted output. Please also note than when it writes the file, it writes three replications of it. We found that the MapReduce job, which does reading, sorting, and writing, takes ***24.47*** times longer to finish for 5 MB. For 10 MB, it is ***42.79*** times. This clearly shows how the write and data transfer operations of a MapReduce job are more expensive than a simple read from only one replica. Because of the number of jobs, the 3-job plan is doing much more disk read and write operations as well as network data transfers and as a result is slower than the 2-job plan even if it is reading less input data.

Because of these reasons, we do not pursue the ideal model. We follow the practical model, which is to generate a query plan having minimum possible jobs. However, while generating a minimum job plan, whenever we need to choose a join to be considered in a job among more than one joins, instead of choosing randomly, we use the summary join statistics. This is described in Section 4.4.

4.3 Query Plan Generation

In this section, first we define the query plan generation problem, and show that generating the best (i.e., least cost) query plan for the ideal model (Section 4.2.1) as well as for the practical model (Section 4.2.2) is computationally expensive. Then, we will present a heuristic and a greedy approach to generate an approximate solution to generate the best plan.

Running example. We will use the following query as a running example in this section:

Listing 5. Running Example

```
SELECT ?V,?X,?Y,?Z WHERE{
?X   rdf:type ub:GraduateStudent
?Y   rdf:type ub:University
?Z   ?V ub:Department
?X   ub:memberOf ?Z
?X   ub:undergraduateDegreeFrom ?Y}
```

In order to simplify the notations, we will only refer to the TPs by the variable in that pattern. For example, the first TP *(?X rdf:type ub:GraduateStudent)* will be represented as simply X. Also, in the simplified version, the whole query would be represented as follows: {X,Y,Z,XZ,XY}. We shall use the notation *join(XY,X)* to denote a join operation between the two TPs XY and X on the common variable X.

Definition 7 (The Minimum Cost Plan Generation Problem). *(Bestplan Problem). For a given query, the Bestplan problem is to generate a job plan so that the total cost of the jobs is minimized. Note that Bestplan considers the more general case where each job has some cost associated with it (i.e., the ideal model).*

Example. Given the query in our running example, two possible job plans are as follows:

Plan 1. $\text{job1} = \text{join}(\text{X,XZ,XY})$, resultant $\text{TPs} = \{\text{Y,Z, YZ}\}$. $\text{job2} = \text{join}(\text{Y, YZ})$, resultant $\text{TPs} = \text{Z, Z}$. $\text{job3} = \text{join}(\text{Z, Z})$. $\text{Totalcost} = \text{cost}(\text{job1}) + \text{cost}(\text{job2}) + \text{cost}(\text{job3})$.

Plan 2. $\text{job1} = \text{join}(\text{XZ, Z})$ and $\text{join}(\text{XY,Y})$ resultant $\text{TPs} = \text{X, X, X}$. $\text{job2} = \text{join}(\text{X, X, X})$. $\text{Totalcost} = \text{cost}(\text{job1}) + \text{cost}(\text{job2})$.

The Bestplan problem is to find the least cost job plan among all possible job plans.

Related terms.

Definition 8 (Joining Variable). *A variable that is common in two or more triple patterns. For example, in the running example query, X,Y,Z are joining variables, but V is not.*

Definition 9 (Complete Elimination). *A join operation that eliminates a joining variable. For example, in the example query, Y can be completely eliminated if we join (XY,Y).*

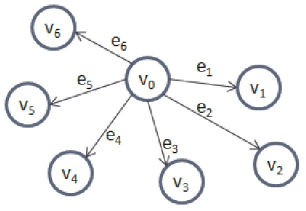
Definition 10 (Partial Elimination). *A join operation that partially eliminates a joining variable. For example, in the example query, if we perform join(XY,Y) and join(X,ZX) in the same job, the resultant triple patterns would be {X,Z,X}. Therefore, Y will be completely eliminated, but X will be partially eliminated. So, the join(X,ZX) performs a partial elimination.*

Definition 11 (E-Count(v)). *E-count(v) is the number of joining variables in the resultant triple pattern after a complete elimination of variable v. In the running example, join(X,XY,XZ) completely eliminates X, and the resultant triple pattern (YZ) has two joining variables Y and Z. So, E-count$(X) = 2$. Similarly, E-count$(Y) = 1$ and E-count$(Z) = 1$.*

4.3.1 Computational Complexity of Bestplan

It can be shown that generating the least cost query plan is computationally expensive, since the search space is exponentially large. At first, we formulate the problem, and then show its complexity.

Problem formulation. We formulate Bestplan as a search problem. Let $G = (V, E)$ be a weighted directed graph, where each vertex $v_i \in V$ represents a state of the triple patterns, and each edge $e_i = (v_{i_1}, v_{i_2}) \in E$ represents a job that makes a transition from state v_{i_1} to state v_{i_2}. v_0 is the initial state, where no joins have been performed, i.e., the given query. Also, v_{goal} is the goal state, which represents a state of the triple pattern where all joins have been performed. The problem is to find the shortest weighted path from v_0 to v_{goal}.



v0={X,Y,Z,XY,XZ}	
v1={X,X,Z,XZ}	e1=(v0,v1)=join(Y,XY)
v2={X,Z,Z}	e2=(v0,v2)=join(Y,XY), join(X,ZX)
v3={X,Y,X,XY}	e3=(v0,v3)=join(Z,XZ)
v4={Y,Y,X}	e4=(v0,v4)=join(Z,XZ),join(X,XY)
v5={X,X,X}	e5=(v0,v5)=join(Z,XZ),join(Y,XY)
v6={YZ,Y,Z}	e6=(v0,v6)=join(X,XY,XZ)

Fig. 4. The (partial) graph for the running example query with the initial state and all states adjacent to it.

For example, in our running example query, the initial state $v_0 = \{X, Y, Z, XY, XZ\}$, and the goal state, $v_{goal} = \phi$, i.e., no more triple patterns left. Suppose the first job (job1) performs join(X,XY,XZ). Then, the resultant triple patterns (new state) would be $v_1 = \{Y, Z, YZ\}$, and job1 would be represented by the edge (v_0, v_1). The weight of edge (v_0, v_1) is the cost of $\text{job1} = \text{cost}(\text{job1})$, where cost is the given cost function. Fig. 4 shows the partial graph for the example query.

Search space size. Given a graph $G = (V, E)$, Dijkstra's shortest path algorithm can find the shortest path from a source to all other nodes in $O(|V|log|V| + |E|)$ time. However, for Bestplan, it can be shown that in the worst case, $|V| \geq 2^K$, where K is the total number of joining variables in the given query. Therefore, the number of vertices in the graph is exponential, leading to an exponential search problem.

Theorem 1. *The worst case complexity of the Bestplan problem is exponential in K, the number of joining variables in the given query.*

Proof. Let us consider the number of possible jobs on the initial query (i.e., number of outgoing edges from v_0). Let n be the maximum number of concurrent complete eliminations possible on the initial query (i.e., maximum number of complete eliminations possible in one job). Any of the $2^n - 1$ combinations of complete eliminations can lead to a feasible job. In our running example, $n = 2$, we can completely eliminate both Y and Z concurrently in one job. However, we may choose among these eliminations in $2^2 - 1$ ways, namely, eliminate only Y, eliminate only Z, and eliminate both Y and Z in one job. $2^2 - 1$ different jobs can be generated. For each combination of complete eliminations, there may be zero or more possible partial eliminations. Again, we may choose any combination of those partial eliminations. For example, if we choose to eliminate Y only, then we can partially eliminate X. We may or may not choose to partially eliminate X, leading to two different job patterns. Therefore, the minimum number of possible jobs (outgoing edges) on the initial query (i.e., v_0) is $2^n - 1$. Note that each of these jobs (i.e., edges) will lead to a unique state (i.e., vertex). Therefore, the number of vertices adjacent to v_0 is at least $2^n - 1$. In the worst case, $n = K$, meaning, the minimum number of vertices adjacent to v_0 is $2^K - 1$. Note that we are not even counting the vertices (i.e., states) that are generated from these $2^K - 1$ vertices. Since the complexity of computing the least cost path from v_0 to v_{goal} is at least $O(|V|log|V| + |E|)$, the solution to the Bestplan problem is exponential in the number of joining variables in the worst case. □

However, we may still solve Bestplan in reasonable amount of time if K is small. This solution would involve generating the graph G and then finding the shortest path from v_0 to v_{goal}.

4.3.2 Relaxed Bestplan Problem and Approximate Solution

In the Relaxed Bestplan problem, we assume uniform cost for all jobs. Although this relaxation does not reduce the search space, the problem is reduced to finding a job plan having the minimum number of jobs. Note that this is the problem for the practical version of the model (Section 4.2.2).

Definition 12 (Relaxed Bestplan Problem). *The Relaxed Bestplan problem is to find the job plan that has the minimum number of jobs.*

Next, we show that if joins are reasonably chosen, and no eligible join operation is left undone in a job, then we may set an upper bound on the maximum number of jobs required for any given query. However, it is still computationally expensive to generate all possible job plans. Therefore, we resort to a greedy algorithm (Algorithm 1), that finds an approximate solution to the Relaxed Bestplan problem, but is guaranteed to find a job plan within the upper bound.

Algorithm 1. Relaxed-Bestplan (Query Q)

```
 1: Q ← Remove_non-joining_variables(Q)
 2: while Q ≠ Empty do
 3:   $J$ ← 1 //Total number of jobs
 4:   $U = \{u_1, \ldots, u_K\}$ ← All variables sorted in
      non-decreasing order of their E-counts
 5:   $Job_J$ ← Empty //List of join operations in the
      //current job
 6:   tmp ← Empty // Temporarily stores resultant
      //triple patterns
 7:   for $i = 1$ to $K$ do
 8:     if Can-Eliminate(Q,$u_i$)=true then
        // complete or partial elimination possible
 9:       tmp ← tmp ∪ Join-result(TP(Q,$u_i$))
10:       Q ← Q - TP(Q,$u_i$)
11:       $Job_J$ ← $Job_J$ ∪ join(TP(Q,$u_i$))
12:     end if
13:   end for
14:   Q ← Q ∪ tmp
15:   $J$ ← $J + 1$
16: end while
17: return $\{Job_1, \ldots, Job_{J-1}\}$
```

Definition 13 (Early Elimination Heuristic). *The early elimination heuristic makes as many complete eliminations as possible in each job.*

This heuristic leaves the fewest number of variables for join in the next job. In order to apply the heuristic, we must first choose the variable in each job with the least E-count. This heuristic is applied in Algorithm 1.

Description of Algorithm 1. The algorithm starts by removing all the nonjoining variables from the query Q. In our running example, $\mathrm{Q} = \{\mathrm{X, Y, VZ, XY, XZ}\}$, and removing the nonjoining variable V makes $\mathrm{Q} = \{\mathrm{X, Y, Z, XY, XZ}\}$. In the while loop, the job plan is generated, starting from Job_1. In line 4, we sort the variables according to their E-count. The sorted variables are: $U = \{Y, Z, X\}$, since Y, and Z have $\text{E-count} = 1$, and X has $\text{E-count} = 2$. For each job, the list of join operations is stored in the variable Job_J, where J is the ID of the current job. Also, a temporary variable tmp is used to store the resultant triples of the joins to be performed in the current job (line 6). In the for loop, each variable is checked to see if the variable can be completely or partially eliminated (line 8). If yes, we store the join result in the temporary variable (line 9), update Q (line 10), and add this join to the current job (line 11). In our running example, this results in the following operations: Iteration 1 of the for loop: $u_1(=\mathrm{Y})$ can be completely eliminated. Here, $\mathrm{TP(Q, Y)} = \text{the}$ triple patterns in Q containing

$$\mathrm{Y} = \{\mathrm{Y, XY}\}.\ \text{Join-result}(\mathrm{TP(Q, Y)}) = \text{Join-result}(\{\mathrm{Y, XY}\}) = \text{resultant}$$

triple after the $\text{join}(\mathrm{Y, XY}) = \mathrm{X}$. So,

$$\begin{aligned} \text{tmp} &= \{\mathrm{X}\}.\mathrm{Q} = \mathrm{Q} - \mathrm{TP(Q, Y)} \\ &= \{\mathrm{X, Y, Z, XY, XZ}\} - \{\mathrm{Y, XY}\} = \{\mathrm{X, Z, XZ}\}. \\ Job_1 &= \{\text{join}(\mathrm{Y, XY})\}. \end{aligned}$$

Iteration 2 of the for loop: $u_2(=\mathrm{Z})$ can be completely eliminated. Here, $\mathrm{TP(Q, Z)} = \{\mathrm{Z, XZ}\}$, and

$$\begin{aligned} &\text{Join-result}(\{\mathrm{Z}, XZ\}) = \mathrm{X}.\ \text{So, tmp} = \{\mathrm{X, X}\}, \\ &\mathrm{Q} = \mathrm{Q} - \mathrm{TP(Q, Z)} = \{\mathrm{X, Z, XZ}\} - \{\mathrm{Z, ZX}\} = \{\mathrm{X}\}, \\ &Job_1 = \{\text{join}(\mathrm{Y, XY}),\ \text{join}(\mathrm{Z, XZ})\}. \end{aligned}$$

Iteration 3 of the for loop: $u_3(=\mathrm{X})$ cannot be completely or partially eliminated, since there is no other TP left to join with it. Therefore, when the for loop terminates, we have $Job_1 = \{\text{join}(\mathrm{Y, XY}), \text{join}(\mathrm{Z, XZ})\}$, and $\mathrm{Q} = \{\mathrm{X, X, X}\}$. In the second iteration of the while loop, we will have $Job_2 = \text{join}(\mathrm{X, X, X})$. Since after this join, Q becomes Empty, the while loop is exited. Finally, $\{Job_1, Job_2\}$ are returned from the algorithm.

Theorem 2. *For any given query Q, containing K joining variables and N triple patterns, Algorithm Relaxed-Bestplan(Q) generates a job plan containing at most J jobs, where*

$$J = \begin{cases} 0, & N = 0, \\ 1, & N = 1 \text{ or } K = 1, \\ \min(\lceil 1.71 \log_2 N \rceil, K), & N, K > 1. \end{cases} \qquad (4)$$

Proof. The first two cases are trivial. For the third case, we need to show that the number of jobs is 1) at most K, and 2) at most $\lceil 1.71 \log_2 N \rceil$. It is easy to show that the number of jobs can be at most K. Since with each job, we completely eliminate at least one variable, we need at most K jobs to eliminate all variables. In order to show that 2) is true, we consider the job construction loop (for loop) of the algorithm for the first job. In the for loop, we try to eliminate as many variables as possible by joining TPs containing that variable. Suppose L TPs could not participate in any join because of conflict between one (or more) of their variables and other triple patterns already taken by another join in the same job. In our running example, TP X could not participate in any join in Job_1 since other TPs containing X have already been taken by other joins. Therefore, $L = 1$ in this example. Note that each of the L TPs had conflict with one (or more) joins in the job. For example, the left-over TP X had conflict with both Join(Y,XY), and Join(Z,ZX). It can be shown that for each of the L TPs, there is at least one unique Join operation which is in conflict with the TP. Suppose there is a TP tp_i, for which it is not true (i.e., tp_i does not have a unique conflicting Join). Therefore, tp_i must be sharing a conflicting Join with another TP tp_j (that is why the Join is not unique for tp_i). Also, tp_i and tp_j do not have any variable in common, since otherwise we could join them, reducing the value of L. Since both tp_i and tp_j are in conflict with the Join, the Join must involve a variable that does not belong to either tp_i or tp_j. To illustrate this with an example, suppose the conflicting Join is join (UX,UV), and $tp_i = \mathrm{X}$, $tp_j = \mathrm{V}$. It is clear that E-count of U must be at least 2, whereas E-count of X and V is 1. Therefore, X and Y must have been considered for elimination before U. In this case, we would have chosen the joins: join(X,UX) and join(V,UV), rather than join (UX,UV). So, either tp_i (and tp_j) must have a unique conflicting Join, or tp_i must have participated in a join.

To summarize the fact, there have been at least $M >= L$ joins selected in Job_1. So, the total number of TPs left after executing all the joins of Job_1 is $M + L$. Note that each of the M joins involves at least two TPs. Therefore, $2M + L \le N$, where N is the total number of TPs in the given query. From the above discussion, we come up with the following relationships:

$$\begin{aligned}
&2M + L \le N \Rightarrow 2(L+\epsilon) + L \le N \quad (\text{Letting } \epsilon \ge 0)\\
&\quad \Rightarrow 3L + 2\epsilon \le N\\
&\quad \Rightarrow 2L + \frac{4}{3}\epsilon \le \frac{2}{3}N \quad (\text{Multiplying both sides with } 2/3)\\
&\quad \Rightarrow 2L + \epsilon \le \frac{2}{3}N \Rightarrow M + L \le \frac{2}{3}N.
\end{aligned}$$

So, the first job, as well as each remaining jobs reduces the number of TPs to at least two third. Therefore, there is an integer J such that

$$\begin{aligned}
&\left(\frac{2}{3}\right)^J N \ge 1 \ge \left(\frac{2}{3}\right)^{J+1} N \Rightarrow \left(\frac{3}{2}\right)^J \le N \le \left(\frac{3}{2}\right)^{J+1}\\
&\quad \Rightarrow J \le \log_{3/2} N = 1.71 \log_2 N \le J + 1.
\end{aligned}$$

So, the total number of jobs, J is also bounded by $\lceil 1.71 \log_2 N \rceil$. □

In most real-world scenarios, we can safely assume that more than 100 triples in a query are extremely rare. So, the maximum number of jobs required with the Relaxed-Bestplan algorithm is at most 12.

Complexity of the Relaxed-Bestplan algorithm. The outer loop (while loop) runs at most J times, where J is the upper bound of the number of jobs. The inner (for) loop runs at most K times, where K is the number of joining variables in the given query. The sorting requires $O(K \log K)$ time. Therefore, the overall complexity of the algorithm is $O(K(J + \log K))$.

4.4 Breaking Ties by Summary Statistics

We frequently face situations where we need to choose a join for multiple join options. These choices can occur when both query plans (i.e., join orderings) require the minimum number of jobs. For example, the query shown in Listing 6 poses such a situation.

Listing 6. Query Having Tie Situation

```
?X rdf:type ub:FullProfessor.
?X ub:advisorOf ?Y.
?Y rdf:type ub:ResearchAssistant.
```

The second triple pattern in the query makes it impossible to answer and solve the query with only one job. There are only two possible plans: we can join the first two triple patterns on X first and then join its output with the last triple pattern on Y or we can join the last two patterns first on Y and then join its output with the first pattern on X. In such a situation, instead of randomly choosing a join variable for the first job, we use join summary statistics for a pair of predicates. We select the join for the first job which is more selective to break the tie. The join summary statistics we use is described in [36].

4.5 MapReduce Join Execution

In this section, we discuss how we implement the joins needed to answer SPARQL queries using MapReduce framework of Hadoop. Algorithm 1 determines the number of jobs required to answer a query. It returns an ordered set of jobs. Each job has associated input information. The Job Handler component of our MapReduce framework runs the jobs in the sequence they appear in the ordered set. The output file of one job is the input of the next. The output file of the last job has the answer to the query.

Listing 7 shows LUBM Query 2, which we will use to illustrate the way we do a join using map and reduce methods. The query has six triple patterns and nine joins between them on the variables X, Y, and Z. Our input selection algorithm selects files *type_GraduateStudent*, *type_University*, *type_Department*, all files having the prefix *memberOf*, all files having the prefix *subOrganizationOf*, and all files having the prefix *underGraduateDegreeFrom* as the input to the jobs needed to answer the query.

Listing 7. LUBM Query 2

```
SELECT ?X, ?Y, ?Z WHERE {
?X rdf:type ub:GraduateStudent.
?Y rdf:type ub:University.
?Z rdf:type ub:Department.
?X ub:memberOf ?Z.
?Z ub:subOrganizationOf ?Y.
?X ub:undergraduateDegreeFrom ?Y}
```

The query plan has two jobs. In job 1, triple patterns of lines 2, 5, and 7 are joined on X and triple patterns of lines 3 and 6 are joined on Y. In job 2, triple pattern of line 4 is joined with the outputs of previous two joins on Z and also the join outputs of job 1 are joined on Y.

The input files of job 1 are *type_GraduateStudent*, *type_University*, all files having the prefix *memberOf*, all files having the prefix *subOrganizationOf*, and all files having the prefix *underGraduateDegreeFrom*. In the *map* phase, we first tokenize the input value which is actually a line of the input file. Then, we check the input file name and, if input is from *type_GraduateStudent*, we output a key-value pair having the subject URI prefixed with *X#* the key and a flag string *GS#* as the value. The value serves as a flag to indicate that the key is of type *GraduateStudent*. The subject URI is the first token returned by the tokenizer. Similarly, for input from file *type_University* output a key-value pair having the subject URI prefixed with *Y#* the key and a flag string *U#* as the value. If the input from any file has the prefix *memberOf*, we retrieve the subject and object from the input line by the tokenizer and output a key-value pair having the subject URI prefixed with *X#* the key and the object value prefixed with *MO#* as the value. For input from files having the prefix *subOrganizationOf*, we output key-value pairs making the object prefixed with *Y#* the key and the subject prefixed with *SO#* the value. For input from files having the prefix *underGraduateDegreeFrom*, we output key-value pairs making the subject URI prefixed with *X#* the key and the object value prefixed with *UDF#* the value. Hence, we make either the subject or the object a map output key based on which we are joining. This is the reason why the object is made the key for the triples from files having the prefix *subOrganizationOf* because the joining variable Y is an object in the triple pattern in line 6. For all other inputs, the subject is made the key because the joining variables X and Y are subjects in the triple patterns in lines 2, 3, 5, and 7.

In the *reduce* phase, Hadoop groups all the values for a single key and for each key provides the key and an iterator to the values collection. Looking at the prefix, we can immediately tell if it is a value for X or Y because of the prefixes we used. In either case, we output a key-value pair using the same key and concatenating all the values to make a string value. So, after this *reduce* phase, join on X is complete and on Y is partially complete.

The input files of job 2 are *type_Department* file and the output file of job 1, *job1.out*. Like the *map* phase of job 1, in the *map* phase of job 2, we also tokenize the input value which is actually a line of the input file. Then, we check the input file name and, if input is from *type_Department*, we output a key-value pair having the subject URI prefixed with *Z#* the key and a flag string *D#* as the value. If the input is from *job1.out*, we find the value having the prefix *Z#*. We make this value the output key and concatenate rest of the values to make a string and make it the output value. Basically, we make the *Z#* values the keys to join on Z.

In the *reduce* phase, we know that the key is the value for Z. The values collection has two types of strings. One has X values, which are URIs for graduate students and also Y values from which they got their undergraduate degree. The Z value, i.e., the key, may or may not be a subOrganizationOf the Y value. The other types of strings have only Y values which are universities and of which the Z value is a suborganization. We iterate over the values collection and then join the two types of tuples on Y values. From the join output, we find the result tuples which have values for X, Y, and Z.

5 RESULTS

In this section, we first present the benchmark data sets with which we experimented. Next, we present the alternative repositories we evaluated for comparison. Then, we detail our experimental setup. Finally, we present our evaluation results.

5.1 Data Sets

In our experiments with SPARQL query processing, we use two synthetic data sets: LUBM [18] and SP2B [34]. The LUBM data set generates data about universities by using an ontology.[20] It has 14 standard queries. Some of the queries require inference to answer. The LUBM data set is very good for both inference and scalability testing. For all LUBM data sets, we used the default seed. The SP2B data set is good for scalability testing with complex queries and data access patterns. It has 16 queries most of which have complex structures.

5.2 Baseline Frameworks

We compared our framework with RDF-3X [29], Jena,[21] and BigOWLIM.[22] RDF-3X is considered the fastest semantic web framework with persistent storage. Jena is an open source framework for semantic web data. It has several models which can be used to store and retrieve RDF data. We chose Jena's in-memory and SDB models to compare our framework with. As the name suggests, the in-memory model stores the data in main memory and does not persist data. The SDB model is a persistent model and can use many off-the-shelf database management systems. We used MySQL database as SDB's backend in our experiments. BigOWLIM is a proprietary framework which is the state-of-the-art significantly fast framework for semantic web data. It can act both as a persistent and nonpersistent storage. All of these frameworks run in a single machine setup.

5.3 Experimental Setup

5.3.1 *Hardware*

We have a 10-node Hadoop cluster which we use for our framework. Each of the nodes has the following configuration: Pentium IV 2.80 GHz processor, 4 GB main memory, and 640 GB disk space. We ran Jena, RDF-3X, and BigOWLIM frameworks on a powerful single machine having 2.80 GHz **quad** core processor, 8 GB main memory, and 1 TB disk space.

5.3.2 *Software*

We used hadoop-0.20.1 for our framework. We compared our framework with Jena-2.5.7 which used MySQL 14.12 for its SDB model. We used BigOWLIM version 3.2.6. For RDF-3X, we utilized version 0.3.5 of the source code.

20. http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl.
21. http://jena.sourceforge.net.
22. http://www.ontotext.com/owlim/big/index.html.

TABLE 4
Comparison with RDF-3X

	LUBM-10,000		LUBM-20,000		LUBM-30,000	
	RDF-3X	HadoopRDF	RDF-3X	HadoopRDF	RDF-3X	HadoopRDF
Query 1	0.373	248.3	0.219	418.523	0.412	591
Query 2	1240.21	801.9	3518.485	924.437	**FAILED**	1315.2
Query 4	0.856	1430.2	0.445	2259.768	1.160	3405.6
Query 9	225.54	1663.4	**FAILED**	3206.439	**FAILED**	4693.8
Query 12	0.298	204.4	0.825	307.715	1.254	425.4
Query 13	380.731	325.4	480.758	462.307	1287	697.2

5.4 Evaluation

In this section, we present performance comparison between our framework, RDF-3X, Jena In-Memory and SDB models, and BigOWLIM.

Table 4 summarizes our comparison with RDF-3X. We used three LUBM data sets: 10,000, 20,000, and 30,000 which have more than 1.1, 2.2, and 3.3 billion triples, respectively. Initial population time for RDF-3X took 655, 1,756, and 3,353 minutes to load the data sets, respectively. This shows that the RDF-3X load time is increasing exponentially. LUBM (30,000) has three times as many triples as LUBM (10,000) yet it requires more than five times as long to load.

For evaluation purposes, we chose LUBM Queries 1, 2, 4, 9, 12, and 13 to be reported in this paper. These queries provide a good mixture and include simple and complex structures, inference, and multiple types of joins. They are representatives of other queries of the benchmark and so reporting only these covers all types of variations found in the queries we left out and also saves space. Query 1 is a simple selective query. RDF-3X is much faster than HadoopRDF for this query. RDF-3X utilizes six indexes [29] and those six indexes actually make up the data set. The indexes provide RDF-3X a very fast way to look up triples, similar to a hash table. Hence, a highly selective query is efficiently answered by RDF-3X. Query 2 is a query with complex structures, low selectivity, and no bound objects. The result set is quite large. For this query, HadoopRDF outperforms RDF-3X for all three data set sizes. RDF-3X fails to answer the query at all when the data set size is 3.3 billion triples. RDF-3X returns memory segmentation fault error messages, and does not produce any query results. Query 4 is also a highly selective query, i.e., the result set size is small because of a bound object in the second triple pattern but it needs inferencing to answer it. The first triple pattern uses the class *Person* which is a superclass of many classes. No resource in LUBM data set is of type *Person*, rather there are many resources which are its subtypes. RDF-3X does not support inferencing so we had to convert the query to an equivalent query having some union operations. RDF-3X outperforms HadoopRDF for this query. Query 9 is similar in structure to Query 2 but it requires significant inferencing. The first three triple patterns of this query use classes of which are not explicitly instantiated in the data set. However, the data set includes many instances of the corresponding subclasses. This is also the query which requires the largest data set join and returns the largest result set out of the queries we evaluated. RDF-3X is faster than HadoopRDF for 1.1 billion triples data set but it fails to answer the query at all for the other two data sets. Query 12 is similar to Query 4 because it is both selective and has inferencing in one triple pattern. RDF-3X beats HadoopRDF for this query. Query 13 has only two triple patterns. Both of them involve inferencing. There is a bound subject in the second triple pattern. It returns the second largest result set. HadoopRDF beats RDF-3X for this query for all data sets. RDF-3X's performance is slow because the first triple pattern has very low selectivity and requires low selectivity joins to perform inference via backward chaining.

These results lead us to some simple conclusions. RDF-3X achieves the best performance for queries with high selectivity and bound objects. However, HadoopRDF outperforms RDF-3X for queries with unbound objects, low selectivity, or large data set joins. RDF-3X cannot execute the two queries with unbound objects (Queries 2 and 9) for a 3.3 billion triples data set. This demonstrates that HadoopRDF is more scalable and handles low selectivity queries more efficiently than RDF-3X.

We also compared our implementation with the Jena In-Memory model and the SDB models and BigOWLIM. Due to space and time limitations, we performed these tests only for LUBM Queries 2 and 9 from the LUBM data set. We chose these queries because they have complex structures and require inference. It is to be noted that BigOWLIM needed **7 GB** of Java heap space to successfully load the billion triples data set. Figs. 5 and 6 show the performance comparison for the queries, respectively. In each of these figures, the X-axis represents the number of triples (in billions) and the Y-axis represents the time (in seconds). We ran BigOWLIM only for the largest three data sets as we are interested in its performance with large data sets. 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 data set grew, Jena In-Memory model ran out of memory space. Our implementation was much faster than Jena SDB model for large data sets. For example, in Fig. 5 for 110 million triples, our approach took 143.5 seconds as compared to about 5,000 seconds for Jena-SDB model. In Fig. 6, we can see that Jena SDB model could not finish answering Query 9. Jena In-Memory Model worked well for small data sets but became slower than our implementation as the data set size grew and eventually ran out of memory.

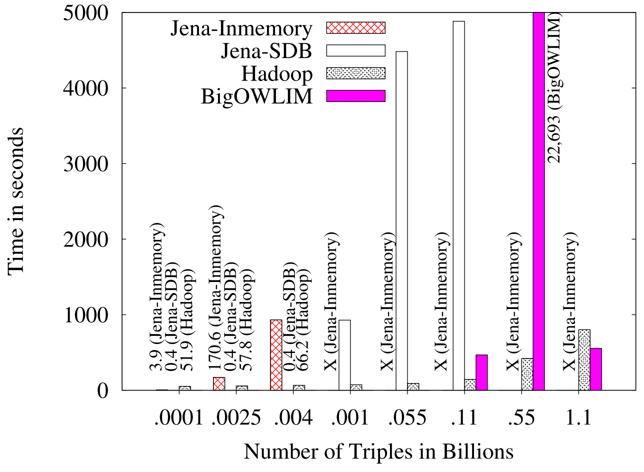


Fig. 5. Response time of LUBM Query 2.

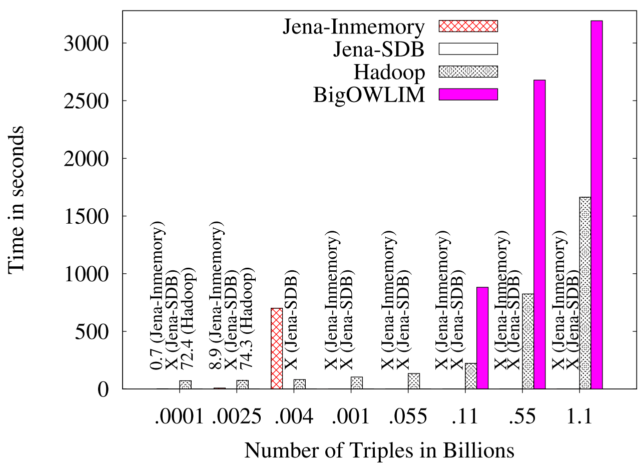


Fig. 6. Response time of LUBM Query 9.

For Query 2 (Fig. 5), BigOWLIM was slower than us for the 110 and 550 million data sets. For 550 million data set, it took **22693.4** seconds, which is abruptly high compared to its other timings. For the billion triple data set, BigOWLIM was faster. It should be noted that our framework does not have any indexing or triple cache whereas BigOWLIM exploits indexing which it loads into main memory when it starts. It may also prefetch triples into main memory. For Query 9 (Fig. 6), our implementation is faster than BigOWLIM in all experiments.

It should be noted that our RDF-3X queries and HadoopRDF queries were tested using cold runs. What we mean by this is that main memory and file system cache were cleared prior to execution. However, for BigOWLIM, we were forced to execute hot runs. This is because it takes a significant amount of time to load a database into BigOWLIM. Therefore, we will always easily outperform BigOWLIM for cold runs. So, we actually tested BigOWLIM for hot runs against HadoopRDF for cold runs. This gives a tremendous advantage to BigOWLIM, yet for large data sets, HadoopRDF still produced much better results. This shows that HadoopRDF is much more scalable than BigOWLIM, and provides more efficient queries for large data sets.

The final tests we have performed are an in-depth scalability test. For this, we repeated the same queries for eight different data set sizes, all the way up to 6.6 billion. Table 5 shows query time to execute the plan generated using *Relaxed-Bestplan* algorithm on different-sized data sets. The first column represents the number of triples in the data set. Columns 2 to 6 of Table 5 represent the five selected queries from the LUBM data set whereas the last three columns are the queries from SP2B data set. Query answering time is in seconds. The number of triples is rounded. As expected, as the number of triples increases, the time to answer a query also increases. For example, Query 1 for 100 million triples took 66.3 seconds whereas for 1,100 million triples 248.3 seconds and for 6.6 billion triples 1253.8 seconds. But we can see that this increase in time is sublinear which is a very good property of our framework. 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. As we determine inferred relations on the fly, queries requiring inference take longer times in our framework. Queries 9 and 12 also require inferencing.

As the size of the data set grows, the increase in time to answer a query does not grow proportionately. The increase in time is always less. For example, there are 10 times as many triples in the data set of 10,000 universities than 1,000 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 data sets. 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.

TABLE 5
Query Runtimes for LUBM and SP2B Data Set

Triples	LUBM-DATASET					SP2B-DATASET		
(Billions)	Q-1	Q-2	Q-4	Q-9	Q-12	Q-1	Q-2	Q-3a
0.11	66.3	143.5	164.4	222.4	67.4	82.7	590.9	85.2
0.55	153.7	421.2	729.1	824.1	135	229.2	2119.4	225.1
1.1	248.3	801.9	1430.2	1663.4	204.4	436.8	4578.7	402.6
2.2	418.5	924.4	2259.7	3206.4	307.7			
3.3	591.4	1315.4	3406.5	4693.8	425.8			
4.4	870.9	1997.3	4401.9	6488	532.5			
5.5	1085.9	2518.6	5574.8	7727.3	650.6			
6.6	1253.8	3057.8	6621	9064.7	830.8			

6 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 these 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, an algorithm to determine a query processing plan, whose worst case is bounded, to answer an SPARQL query and a simplified cost model to be used by the algorithm. Our experiments demonstrate that our system is highly scalable. If we increase the data volume, the delay introduced to answer a query does not increase proportionally. The results indicate that for very large data sets (over one billion triples), HadoopRDF is preferable and more efficient if the query includes low selectivity joins or significant inference. Other solutions may be more efficient if the query includes bound objects which produce high selectivity.

In the future, we would like to extend the work in few directions. First, we will investigate more sophisticated query model. We will cache statistics for the most frequent queries and use dynamic programming to exploit the statistics. Second, we will evaluate the impact of the number of reducers, the only parameter of a Hadoop job specifiable by user, on the query runtimes. Third, we will investigate indexing opportunities and further usage of binary formats. Finally, we will handle more complex SPARQL patterns, e.g., queries having OPTIONAL blocks.

Acknowledgments

This material is based upon work supported by the AFOSR under Award No. FA9550-08-1-0260 and NASA under Award No. 2008-00867-01.

References

[1] D.J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 3-12, Mar. 2009.

[2] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach, "SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management," *VLDB J.*, vol. 18, no. 2, pp. 385-406, Apr. 2009.

[3] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," *Proc. 33rd Int'l Conf. Very Large Data Bases*, 2007.

[4] A. Abouzeid, K. Bajda-Pawlikowski, D.J. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *Proc. VLDB Endowment*, vol. 2, pp. 922-933, 2009.

[5] M. Atre, J. Srinivasan, and J.A. Hendler, "BitMat: A Main-Memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries," *Proc. Int'l Semantic Web Conf.*, 2008.

[6] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 479-490, 2006.

[7] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations," *Proc. 13th Int'l World Wide Web Conf. Alternate Track Papers and Posters*, pp. 74-83, 2004.

[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proc. Seventh USENIX Symp. Operating System Design and Implementation*, Nov. 2006.

[9] A. Chebotko, S. Lu, and F. Fotouhi, *Semantics Preserving SPARQL-to-SQL Translation*, Technical Report TR-DB-112007-CLF, 2007.

[10] E.I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An Efficient SQL-Based RDF Querying Scheme," *Proc. Int'l Conf. Very Large Data Bases (VLDB '05)*, 2005.

[11] 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," *Proc. Neural Information Processing Systems (NIPS)*, 2007.

[12] R. Cyganiak, *A Relational Algebra for SPARQL*, Technical Report HPL-2005-170, 2005.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation*, 2004.

[14] L. Ding, T. Finin, Y. Peng, P.P. da Silva, and D.L. Mcguinness, "Tracking RDF Graph Provenance Using RDF Molecules," *Proc. Fourth Int'l Semantic Web Conf.*, 2005.

[15] R. Elmasri and B. Navathe, *Fundamentals of Database Systems*. Pearson Education, 1994.

[16] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-Store Support for RDF Data Management: Not All Swans Are White," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1553-1563, Aug. 2008.

[17] Y. Guo and J. Heflin, "A Scalable Approach for Partitioning OWL Knowledge Bases," *Proc. Second Int'l Workshop Scalable Semantic Web Knowledge Base Systems*, 2006.

[18] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, pp. 158-182, 2005.

[19] Y. Guo, Z. Pan, and J. Heflin, "An Evaluation of Knowledge Base Systems for Large OWL Datasets," *Proc. Int'l Semantic Web Conf.*, 2004.

[20] M.F. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce," *Proc. First Int'l Conf. Cloud Computing*, http://www.utdallas.edu/mfh062000/techreport1.pdf, 2009.

[21] M.F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools," *Proc. IEEE Int'l Conf. Cloud Computing*, pp. 1-10, July 2010.

[22] A. Kiryakov, D. Ognyanov, and D. Manov, "OWLIM: A Pragmatic Semantic Repository for OWL," *Proc. Int'l Workshop Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2005.

[23] J.P. Mcglothlin and L.R. Khan, "RDFKB: Efficient Support for RDF Inference Queries and Knowledge Management," *Proc. Int'l Database Eng. and Applications Symp. (IDEAS)*, 2009.

[24] J.P. McGlothlin and L. Khan, "Materializing and Persisting Inferred and Uncertain Knowledge in RDF Datasets," *Proc. AAAI Conf. Artificial Intelligence*, 2010.

[25] A.W. Mcnabb, C.K. Monson, and K.D. Seppi, "MRPSO: MapReduce Particle Swarm Optimization," *Proc. Ann. Conf. Genetic and Evolutionary Computation (GECCO)*, 2007.

[26] P. Mika and G. Tummarello, "Web Semantics in the Clouds," *IEEE Intelligent Systems*, vol. 23, no. 5, pp. 82-87, Sept./Oct. 2008.

[27] J.E. Moreira, M.M. Michael, D. Da Silva, D. Shiloach, P. Dube, and L. Zhang, "Scalability of the Nutch Search Engine," *Proc. 21st Ann. Int'l Conf. Supercomputing (ICS '07)*, pp. 3-12, June 2007.

[28] C. Moretti, K. Steinhaeuser, D. Thain, and N. Chawla, "Scaling Up Classifiers to Cloud Computers," *Proc. IEEE Int'l Conf. Data Mining (ICDM '08)*, 2008.

[29] T. Neumann and G. Weikum, "RDF-3X: A RISC-Style Engine for RDF," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 647-659, 2008.

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

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2008.

[32] P. Pantel, "Data Catalysis: Facilitating Large-Scale Natural Language Data Processing," *Proc. Int'l Symp. Universal Comm.*, 2007.

[33] K. Rohloff, M. Dean, I. Emmons, D. Ryder, and J. Sumner, "An Evaluation of Triple-Store Technologies for Large Data Stores," *Proc. OTM Confederated Int'l Conf. On the Move to Meaningful Internet Systems*, 2007.

[34] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL Performance Benchmark," *Proc. 25th Int'l Conf. Data Eng. (ICDE '09)*, 2009.
[35] Y. Sismanis, S. Das, R. Gemulla, P. Haas, K. Beyer, and J. McPherson, "Ricardo: Integrating R and Hadoop," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2010.
[36] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation," *WWW '08: Proc. 17th Int'l Conf. World Wide Web*, 2008.
[37] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-Store: A Column-Oriented DBMS," *VLDB '05: Proc. 31st Int'l Conf. Very Large Data Bases*, pp. 553-564, 2005.
[38] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, "Scalable Distributed Reasoning Using MapReduce," *Proc. Int'l Semantic Web Conf.*, 2009.
[39] J. Wang, S. Wu, H. Gao, J. Li, and B.C. Ooi, "Indexing Multi-Dimensional Data in a Cloud System," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2010.
[40] J. Weaver and J.A. Hendler, "Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples," *Proc. Eighth Int'l Semantic Web Conf.*, 2009.
[41] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1008-1019, 2008.
[42] Y. Xu, P. Kostamaa, and L. Gao, "Integrating Hadoop and Parallel DBMs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2010.

Mohammad Farhan Husain is working as a software development engineer at Amazon.com. He joined the PhD program in the Department of Computer Science at the University of Texas at Dallas in Fall, 2006. His thesis topic was storage and retrieval of large semantic web graphs in an efficient and scalable manner using cloud computing tools. He graduated in May, 2011.

James McGlothlin received the BSc degree from Vanderbilt University and has extensive industry experience with Dell and IBM. He received the master's degree in 2010 with a 4.0 GPA and expects to receive the PhD degree in 2011. He is working toward the doctoral degree in computer science at the University of Texas at Dallas (UTD). He is the inventor on six software patents. During his tenure with UTD, he has published seven conference papers and one journal paper, all in the fields of semantic web repositories, RDF graph storage, and cloud computing.

Mohammad Mehedy Masud graduated from Bangladesh University of Engineering and Technology with BS and MS degrees in computer science and engineering in 2001 and 2004, respectively. He received the PhD degree from University of Texas at Dallas (UTD) in December 2009. He is a postdoctoral research associate at the UTD. His research interests are in data stream mining, machine learning, and intrusion detection using data mining. His recent research focuses on developing data mining techniques to classify data streams. He has published more than 20 research papers in journals including *IEEE Transactions on Knowledge and Data Engineering*, and peer-reviewed conferences including ICDM, ECML/PKDD, and PAKDD. He is also the lead author of the book titled *Data Mining Tools for Malware Detection*, and the principal inventor of US Patent Application titled "Systems and Methods for Detecting a Novel Data Class."

Latifur R. Khan received the BSc degree in computer science and engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in November 1993. He received the MS and PhD degrees in computer science from the University of Southern California in December 1996 and August 2000, respectively. He is currently an associate professor in the Computer Science Department at the University of Texas at Dallas (UTD), where he has been teaching and conducting research since September 2000. His research work is supported by grants from NASA, the Air Force Office of Scientific Research (AFOSR), US National Science Foundation (NSF), the Nokia Research Center, Raytheon, CISCO, Tektronix. In addition, he is the director of the state-of-the-art DBL@UTD, UTD Data Mining/Database Laboratory, which is the primary center of research related to data mining, semantic web, and image/video annotation at University of Texas-Dallas. His research areas cover data mining, multimedia information management, semantic web, and database systems with the primary focus on first three research disciplines. He has served as a committee member in numerous prestigious conferences, symposiums, and workshops. He has published more than 150 papers in prestigious journals and conferences.

Bhavani Thuraisingham is the Louis A. Beecherl, Jr. I distinguished professor in the Erik Jonsson School of Engineering and Computer Science (CS) at the University of Texas at Dallas (UTD) since September 2010. She joined UTD in October 2004 as a professor of computer science and the director of the Cyber Security Research Center (CSRC). She is the recipient of the IEEE CS 1997 Technical Achievement Award, the 2010 Research Leadership Award presented by the IEEE ITS and IEEE SMC, and the 2010 ACM SIGSAC Outstanding Contributions Award. She has more than 30 years experience in the commercial industry (Control Data, Honeywell), MITRE, US National Science Foundation (NSF) and Academia, and has led several research projects. Her work has resulted in more than 100 journal articles, more than 200 conference papers, three US patents, and 10 books. She is an elected fellow of the IEEE, the AAAS, and the British Computer Society.