Scalable Complex Query Processing Over Large Semantic Web Data Using Cloud

Mohammad Farhan Husain*, James McGlothlin†, Latifur Khan‡ and Bhavani Thuraisingham§
Department of Computer Science
University of Texas at Dallas
800 West Campbell Road, Richardson, TX 75080-3021
*mfh062000@utdallas.edu
†jpm083000@utdallas.edu
‡lkhan@utdallas.edu
§bhavani.thuraisingham@utdallas.edu

***Abstract*—Cloud computing solutions continue to grow increasingly popular both in research and in the commercial IT industry. With this popularity comes ever increasing challenges for the cloud computing service providers. Semantic web is another domain of rapid growth in both research and industry. RDF datasets are becoming increasingly large and complex and existing solutions do not scale adequately. In this paper, we will detail a scalable semantic web framework built using cloud computing technologies. We define solutions for generating and executing optimal query plans. We handle not only queries with Basic Graph Patterns (BGP) but also complex queries with optional blocks. We have devised a novel algorithm to handle these complex queries. Our algorithm minimizes binding triple patterns and joins between them by identifying common blocks by algorithms to find subgraph isomorphism and building a query plan utilizing that information. We utilize Hadoop's MapReduce framework to process our query plan. We will show that our framework is extremely scalable and efficiently answers complex queries.**

***Index Terms*—RDF; Hadoop; Cloud; Semantic Web;**

I. Introduction

As large enterprises look to cut the costs of data infrastructure, cloud computing becomes increasingly popular. Cloud computing technologies can use commodity class hardware to manage and retrieve large amounts of data, creating solutions that are affordable. Similarly, the semantic web is increasingly popular as the framework for exchanging knowledge efficiently. The semantic web technologies are governed by the World Wide Web Consortium (W3C). The most prominent standards are Resource Description Framework[1] (RDF) and SPARQL Protocol and RDF Query Language[2] (SPARQL). RDF is the standard for storing and representing data and SPARQL is a query language to retrieve data from an RDF store. The power of these Semantic Web technologies can be successfully harnessed in a cloud computing environment to provide the user with capability to efficiently store and retrieve data for data intensive applications.

Scalability is the predominant challenge in the semantic web as datasets continue to grow larger, and more datasets are integrated and linked. RDF graphs are becoming huge and graph patterns are becoming more complex. At present, existing semantic web frameworks are not sufficiently scalable. A cloud computing solution can be built to overcome these scalability and performance problems.

Companies pioneering cloud computing such as Salesforce.com and Amazon have platforms such as EC2[3], S3[4], Force.com[5] etc. These are proprietary, closed source platforms. However, Hadoop[6] is an emerging Cloud Computing tool which is open source and supported by Amazon, the leading Cloud Computing hosting company. It is a distributed file system where files can be saved with replication, and would be an ideal candidate for building a storage system. Hadoop features high fault tolerance and great reliability. In addition, it also contains an implementation of the MapReduce [6] programming model, a functional programming model which is suitable for the parallel processing of large amounts of data. By partitioning data into a number of independent chunks, MapReduce processes run on these chunks, making parallelization easier.

In this paper, we will describe our cloud computing solution for RDF graphs. We will describe a schema to store RDF data in Hadoop and a query answering system utilizing MapReduce. We will show through experiments with popular benchmark that our solution does achieve exceptional scalability and performance.

SPARQL queries with OPTIONAL blocks pose significant challenge while generating a query answering plan. Such queries have more than one triple pattern[7] blocks, potentially with triple patterns repeated in more than one block. Listing 1 shows an example query with an optional block. The query retrieves names of resources and optionally ages of the corresponding resource if available. Table II shows the result of the query when run on the data shown Table I.

[1]http://www.w3.org/TR/rdf-primer
[2]http://www.w3.org/TR/rdf-sparql-query
[3]http://aws.amazon.com/ec2
[4]http://aws.amazon.com/s3
[5]http://www.salesforce.com/platform
[6]http://hadoop.apache.org
[7]http://www.w3.org/TR/rdf-sparql-query/#sparqlTriplePatterns

978-0-7695-4460-1/11 $26.00 © 2011 IEEE
DOI 10.1109/CLOUD.2011.15

Subject	Object	Predicate
http://utd.edu/s1	:name	”John Smith”
http://utd.edu/s1	:age	”24”
http://utd.edu/s2	:name	”John Doe”
http://utd.edu/s2	:age	”32”
http://utd.edu/s3	:name	”Jane Doe”

TABLE I
EXAMPLE DATA

?name	?age
”John Smith”	”24”
”John Doe”	”32”
”John Doe”	”Jane Doe”

TABLE II
EXAMPLE QUERY RESULT

Listing 1. Example query with OPTIONAL block

```
SELECT ?name, ?age, WHERE {
?x :name ?name . OPTIONAL { ?x :age ?age .}}
```

The resources *http://utd.edu/s1* and *http://utd.edu/s2* have both name and age whereas the resource *http://utd.edu/s3* does not have any age. But because the age is optional, the name of *http://utd.edu/s3* is also part of the result even though its age is not there in the data set.

In [8], we first introduced our novel scheme to store RDF data in the Hadoop Distributed File System and our algorithm for answering SPARQL queries having only BGP using Hadoop. This paper expands on that research. We will present a novel solution for handling queries with optional blocks. This allows queries to be efficiently answered that could not be answered at all by our previous version. Our previously reported algorithms can only handle SPARQL queries with only Basic Graph Patterns[8] (BGP). We will also introduce a query rewriting algorithm. It can rewrite a query to an equivalent simpler form which requires less time to answer than the original one. We do this by leveraging our schema and the ontology of the dataset, if available. We can rewrite the query even if no ontology is available for the dataset. We also perform more complete and thorough experimental analysis. This includes experimental results with the SP^{2}B benchmark dataset [18] and new queries from its benchmark query set.

The remainder of this paper is organized as follows: in Section II, we discuss relevant research. In Section III, we discuss the system architecture and our data storage scheme. In Section IV, we discuss how we answer a SPARQL query. In Section IV-D3, we discuss our approach to handle queries with OPTIONAL blocks. In Section V, we present the results of our experiments. Finally, in Section VI, we draw some conclusions and discuss probable areas which we have identified for improvement in the future.

II. RELATED WORKS

MapReduce is a new technology that is rapidly growing in popularity and usage. MapReduce solutions are being implemented in a wide variety of fields both within and outside of the web. Within the web domain, Google uses MapReduce for web indexing, data storage and social networking [3]. Yahoo! uses MapReduce extensively in their data analysis tasks [15]. IBM has successfully experimented with a scale-up scale-out search framework using MapReduce technology [11].

MapReduce has also been used to develop solutions for data mining [12]. In [4], researchers rewrite well-known machine learning algorithms to take advantage of multicore machines by leveraging the MapReduce programming paradigm.

In [2], the authors propose to combine MapReduce with existing relational database techniques. In [1], researchers reported a vertically partitioned DBMS for storage and retrieval of RDF data. They observed performance improvement with their scheme over traditional relational database schemes. Researchers have also used MapReduce for inferencing RDF data. In [22], they have proposed a solution to do scalable distributed reasoning of RDF data using MapReduce. The solution proposed in [23] can also be implemented by MapReduce.

In the semantic web arena, there has not been much work done with MapReduce technology.The BioMANTA[9] project performed some research working with large RDF data storage in Hadoop. They proposed extensions to RDF Molecules and implemented a MapReduce based Molecule store [14]. They used MapReduce to answer the queries. They have queried a maximum of 4 million triples. Our work is much more scalable than their solution; we have queried billions of triples. Also, our storage schema is specifically designed 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.

We have multiple algorithms to determine the best query processing plan to answer a query based on a cost model or heuristics. By generating such a plan, we can determine the input files of a job and the order in which they should be run. To the best of our knowledge, we are the first to come up with a storage schema for RDF data using flat files in HDFS, and a MapReduce job determination algorithm to answer a SPARQL query.

There are many research projects proposing solution for storing and querying RDF graphs. Historically, the most popular such solution has been Jena. The Jena framework does offer solutions for inference and optional blocks. However, Jena is not able to provide scalability or query efficiency that is even competitive with out solution.

RDF-3X [13] is widely considered the fastest semantic web repository. RDF-3X uses query planning, histograms, summary statistics, and other techniques to improve the performance of queries. For some queries, particularly aggregation queries or queries with objects bound to literals, RDF-3X provides optimal efficiency. However, RDF-3X is not able to even answers queries of billions of triples with low selectivity joins. In our previous research [9], we have shown that we

[8]http://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns

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

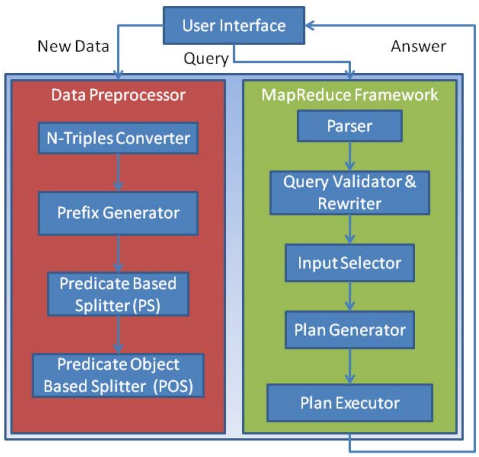

Fig. 1. The System Architecture

outperform RDF-3X for queries involving unbound objects, inference and/or low selectivity joins.

In our previous works [8], [9], we have proposed an exhaustive and a heuristics based algorithm to generate a query processing plan for SPARQL queries. Those algorithms have the limitation that they can only handle queries having simple BGPs. In this work, we propose an algorithm which can handle queries having $OPTIONAL$ blocks. We also propose a query rewriting algorithm which can rewrite many queries to a simpler and efficient to answer equivalent form.

III. Proposed Architecture

Our framework consists of two major components. The left part of Figure 1 depicts the data preprocessing component, and the right part shows the one which answers a query.

We have four subcomponents for data generation and preprocessing. We convert data in RDF/XML[10] or any other format to N-Triples[11] serialization format using our N-Triples Converter component. The Prefix Generator component replaces common prefixes with smaller strings. We save a lot of space by getting rid of these long prefixes. Usually there is not a lot of common prefixes in a data set. The PS component takes the N-Triples data and splits it into predicate files. The predicate based files are then fed into the POS component which splits the predicate files into smaller files based on the type of objects. These steps are described in Section III-B, III-C and III-D.

Our MapReduce framework has four major subcomponents. It takes the SPARQL query from the user and rewrites it, if possible, using the Query Validator and Rewriter component. This component leverages our schema (see Section III-B) using our query rewriting algorithm. The algorithm is presented in Section IV-A. The query rewriting algorithm passes the rewritten or, in case rewriting is not possible, the original query to the Input Selector (see Section IV-B) and Plan Generator. This component selects the input files, by using our algorithm described in Section IV-D, decides how many MapReduce jobs are needed and passes the information to the Join Executer component which runs the jobs using MapReduce framework.

A. Data Generation and Storage

For the experiments of this work, we used the SP^{2}B [18] benchmark data set. It has a data generator which generates DBLP[12] like data in N-Triples serialization format. We do not need to do any conversion with this dataset because this is the format we use for later steps of the preprocessing phase. However, our N-Triples converter is there to convert the input data to N-Triples in case it is not in that format. The following sections talk about the details.

B. File Organization

In Hadoop, a file is the smallest unit of input to a MapReduce job and in absence of caching, a file is always read from the disk each time a job is set up. Hence, to reduce the input size of our jobs, we divide the data into multiple smaller files. The splitting is done in two steps which we discuss in the following sections.

C. Predicate Split (PS)

At first, We divide the data according to the predicates. In real world RDF datasets, the number of distinct predicates is not very large [19]. This division immediately enables us to cut down the search space for any query which does not have a variable[13] 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*.

D. Predicate Object Split (POS)

1) Split Using Explicit Type Information of Object: In the next step, we work with the *rdf_type* file. The file is divided into as many files as the number of distinct objects the *rdf:type* predicate has. For example, if in the ontology the leaves of the class hierarchy are $c_1, c_2, ..., c_n$ then we will create files for each of these leaves and the file names will be like *rdf-type_*c_1, *rdf-type_*c_2, ... , *rdf-type_*c_n. As the object values $c_1, c_2, ..., c_n$ are no longer needed to be stored inside the file, we further reduces the amount of space needed to store the data.

2) Split Using Implicit Type Information of Object: In the final step, the remaining predicate files are divided 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. The type information of a URI object can be retrieved from the *rdf-type_** files. The URI objects move into their respective file named as *predicate_type*. For example, if a triple has the predicate p and the type of the URI object is c_i, then the subject and object appears in one line in the file p_c_i. To do this division we need to join a predicate file with the *rdf-type_** files to retrieve the type information.

[10] http://www.w3.org/TR/rdf-syntax-grammar
[11] http://www.w3.org/2001/sw/RDFCore/ntriples
[12] http://www.informatik.uni-trier.de/ ley/db
[13] http://www.w3.org/TR/rdf-sparql-query/#sparqlQueryVariables

IV. MapReduce Framework

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

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

A. Query Rewriting

When a query is submitted by the user, we sometimes can take advantage of our schema and rewrite the query in equivalent simpler form. If a variable has its type information in a triple pattern having that variable as the subject, the predicate *rdf:type* and a bound object, which is the type of the variable, we can eliminate this triple pattern if the variable is used as an object in some other triple pattern. We can do this because, as described earlier, we divide the predicate files according to the type of the objects. So if, for the triple pattern which has the variable as object, we choose the predicate file having that specific type of objects as input, we do not need the triple pattern having the type information. An example illustrates it better. Listing 2 shows LUBM query 2 and Listing 3 shows its rewritten form. Both the variables Y and Z have type information in the second and third triple patterns. They are also used as objects in the last three triple patterns. If we include the files *ub:memberOf_Department*, *ub:subOrganizationOf_University* and *ub:undergraduateDegreeFrom_University* in the input file set for the last triple patterns respectively, we can guarantee that the values bound to the variables Y and Z in the query result would be of type *ub:University* and *ub:Department* respectively. This is possible because of the way we divide our data (see Section III-D). The rewritten query has two less triple patterns. As a result, it got rid of two joins on the variable Y and two on the variable Z. This rewritten query runs significantly faster than the original query because of having four less joins. Here the rewritten query has predicates which serve as a filename for our input selection phase described in Section IV-B. Note that this type of rewriting is possible even if there is no ontology associated with the dataset. This type of rewriting is very useful because we observed that most of the SPARQL queries have triple patterns with *rdf:type* predicates describing the type of a variable. For example, all the queries of LUBM dataset and all but two queries of SP^{2}B dataset have at least one such triple pattern.

Listing 2. Original 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 }
```

Listing 3. Rewritten LUBM Query 2

```
SELECT ?X, ?Y, ?Z WHERE {
?X rdf:type ub:GraduateStudent .
?X ub:memberOf_Department ?Z .
?Z ub:subOrganizationOf_University ?Y .
?X ub:undergraduateDegreeFrom_University ?Y}
```

However, if we have an ontology available we can do similar rewriting by exploiting the range information of a predicate which might result in further simplification. For example, Listing 4 shows a slightly modified version of LUBM query 2. It does not have the triple pattern having the type information of the variable Y. However, the LUBM ontology says that the range of the predicate $undergraduateDegreeFrom$ is $University$. Hence, we can rewrite the query as Listing 5. Here we reduce the input size because we can include $subOrganizationOf_University$ file in the input set instead of all the files having the prefix $subOrganizationOf$. Note that by getting rid of triple patterns we are reducing the number of joins, not reordering them.

Listing 4. Original Example Query

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

Listing 5. Rewritten Example Query

```
SELECT ?X, ?Y, ?Z WHERE {
?X rdf:type ub:GraduateStudent .
?X ub:memberOf_Department ?Z .
?Z ub:subOrganizationOf_University ?Y .
?X ub:undergraduateDegreeFrom_University ?Y }
```

Hence, for queries, having type information of a variable used as an object in any triple pattern and predicates having range information, we can have rewritten queries which reduces input size and may eliminate few joins. Both of these have significant impact on query runtime.

We also do a trivial optimization by substituting a variable with a bound value found in a $FILTER$[14] clause for that variable. For example, Listing 6 shows SP^{2}B query 3a. We can rewrite it as Listing 7. In this case, we can get rid of the $FILTER$ because once the $?property$ variable is substituted, the $FILTER$ becomes useless.

Listing 6. SP^{2}B Query 3a

[14]http://www.w3.org/TR/rdf-sparql-query/#scopeFilters

```
SELECT ?article WHERE {
?article rdf:type bench:Article .
?article ?property ?value
FILTER (?property=swrc:pages) }
```

Listing 7. Rewritten SP^{2}B Query 3a

```
SELECT ?article WHERE {
?article swrc:pages_Article ?value }
```

B. Input Files Selection

Before determining the jobs, we select the files that need to be inputted to the jobs. We take the query submitted by the user or, if possible, a rewritten one and iterate over the triple patterns. For each triple pattern there are several cases we need to consider. If in a triple pattern, both the predicate and object are variables and the object has not type information available or if the object is concrete, we select all the files in the dataset as input to the jobs and terminate the iteration. If the predicate is a variable but the object has type information available, we select all predicate files having object of that type and add them to the input file set. If the predicate is concrete but the object is variable without any type information, we add all files for the predicate to the input set. If the query has the type information of the object, we add the predicate file which has objects of that type to the input set. If a type associated with a predicate is not a leaf in the ontology tree, we add all subclasses which are leaves in the subtree rooted at the type node in the ontology.

C. Cost Estimation for Query Processing

We run Hadoop jobs to answer a SPARQL query and in many cases more than one job may be needed. Therefore, to generate an optimal plan involving more than one job, we need to estimate cost of the plan. We have discussed this in details in our previous work in [8].

D. Query Plan Generation

Listing 8. 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>}
```

As we discussed earlier, there may be more than one job necessary to answer a query. Hence, there may be multiple ways a SPARQL query can be answered. These ways are typically called query plans. Each of the possible plans to answer a query has different performance in terms of time and space. So, there is a need to find the plan which would provide the best performance. An example can illustrate the issue better. Listing 8 shows LUBM query 12. Recall that a triple pattern cannot take part in two joins on two variables in a job. So the query can not be answered by a single job because of the third triple pattern (?X ub:worksFor ?Y). We can see that there are multiple ways to answer the query. One plan can be to join triple patterns 1 (?X rdf:type ub:Chair) and 3 (?X ub:worksFor ?Y) on variable X in first job, join triple patterns 2 (?Y rdf:type ub:Department) and 4 (?Y ub:subOrganizationOf ¡http://www.U0.edu¿) on variable Y in second job and join the outputs of these two Jobs on variable Y in the third and final job. Another plan can be to join triple patterns 2, 3 and 4 on variable Y in first Job and join triples pattern 1 and the output of first job on variable X in the second and final job. Intuitively, we can say that the more the number of jobs, the slower the answering time is. This is because a job has a setup time and it reads and writes data from and to disk several times. So the first plan is slower than the second one. Hence we can see that there is a need to determine the best query plan before answering a SPARQL query.

2) Heuristics Based Plan Generation: There are a couple of approaches to generate a query plan for a SPARQL query: greedy approach, exhaustive approach [8] etc. The greedy approach is a simple one and generates a plan very quickly. At each step, it selects a variable on which maximum number of joins can be done. But it is not guaranteed that this approach will always generate the best plan. The exhaustive approach generates the best plan by enumerating each possible plan. It uses a cost function to determine the best plan. However, as the search space is exponential in size, the runtime of the algorithm is also exponential. Our latest algorithm, *Relaxed-Bestplan*, is another greedy one which uses a heuristic [9]. In our heuristics based algorithm, we consider only the variables which take part in joins. At each step we determine the elimination count for each variable, i.e. the number of variables left in the triple patterns left in the resultant triple pattern after a join is done on that variable. Then we sort the variables in ascending order of their elimination counts and choose as many full or partial joins as possible. This results in a plan having the least number of jobs. The details of the algorithm can be found in [9].

3) Queries with Optional Blocks: We devise an algorithm which can generate efficient plans for queries with OPTIONAL blocks. Our approach to handle these queries is to build individual plans for each block and answer the query with the help of an operator graph. The concept of operator graph is not new. [17], [20] discuss a similar operator graph approach and there are many others in the literature. For example, the example query we show in Listing 1 would produce the operator graph shown in Figure 2 (a). The figure shows a simple operator graph which has a join operator. $Block1$ consists of the first triple pattern of the query (Listing 1) and $Block2$ consists of the triple pattern in the optional block. The join operator in the graph does a right outer join.

However, the example query shows in Listing 1 is too simple. Real world queries tend to be more complex than that. One such variation is queries having optional blocks with filters. Listing 9 shows a modified version of the example query. It has a filter in the optional block which basically gets rid of the second tuple of the result set shown in Table II. Figure 2 (b) shows the operator graph for this query.

The difference between the operator graph shown in Figure 2 (a) and the one in Figure 2 (b) is the filter below $Block2$.

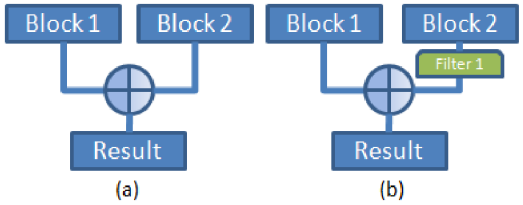

Fig. 2. Operator Graph for (a) Example Query, (b) Modified Example Query

The join operator does right outer join as before. However, we observed that the OPTIONAL blocks frequently repeats triple patterns found in the BGP of the query and there might be filters in the optional blocks which relate to the BGP by using a variable found there. For example, the SP^{2}B Query 6 shown in Listing 10 has such an OPTIONAL block.

Listing 9. Example query having OPTIONAL block with FILTER

```
SELECT ?name, ?age, WHERE { ?x :name ?name .
OPTIONAL { ?x :age ?age . FILTER (?age < 30)}}
```

Listing 10. SP^{2}B Query 6

```
SELECT ?yr ?name ?document WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2)) }
```

We can see that the OPTIONAL block repeats 4 of the 5 triple patterns found in the BGP. If we bind values for both the blocks separately, we would be reading the same data twice off the file system. This is certainly not desirable. To handle this type of query efficiently, first we need to identify the common group of triple patterns, which we call common blocks from now on. Identifying the common blocks is actually the well known problem of subgraph isomorphism. Subgraph isomorphism is proved to be NPC long time ago [5]. Research has already been done to solve this problem in most efficient manner. Ullmann (1976) describes a recursive backtracking procedure for solving the problem in [21] which is deemed to be the most efficient solution found so far. In general, the running time of the algorithm is exponential but in some cases, the running time reduces to linear time. We use Ullmann's algorithm as a subroutine in our algorithm. Algorithm 1 shows the pseudo-code of our algorithm:

Algorithm 1 parses and separates the blocks in line 1. Lines 3 to 10 find common blocks between all possible pairs of the blocks and build a collection of common blocks. Line 5 uses Ullmann's algorithm as a subroutine to find the common block between two blocks. Lines 11 and 12 gets rid of duplicates from the common blocks collection. Once we find the common blocks, we build an operator graph, in lines 13 and 14, which

Algorithm 1 GENERATEPLAN*(query)*

```
1:  blocks ← getBlocks(query)
2:  commonBlocks ← ϕ
3:  for i = 1 to |blocks| do
4:    for j = i + 1 to |blocks| do
5:      commonBlock ← ullmannSI(blocks[i], blocks[j])
6:      if notEmpty(commonBlock) then
7:        commonBlocks ← ∪commonBlock
8:      end if
9:    end for
10: end for
11: uniqueBlocks ←
12: getUniqueBlocks(query, commonBlocks)
13: operatorGraph ←
14: getOperatorGraph(query, commonBlocks)
15: for i = 1 to |uniqueBlocks| do
16:   uniqueBlockPlans ←
17:   Relaxed − Bestplan(uniqueBlocks[i])
18: end for
19: plan ←
20: generatePlan(operatorGraph, uniqueBlockPlans)
21: return plan
```

shows the interactions between the common blocks and how we will get the result using joins and other necessary operators (e.g. Filters). Once the operator graph is built, we find query plans for the common blocks using our Relaxed-Bestplan algorithm in lines 15 to 18. Finally, by combining the query plans for each block with the operator graph in lines 19 and 20, we get the complete plan for the whole query.

An example illustrates the details better. Figure 3 (a) shows the operator graph we build for SP^{2}B Query 6 shown in Listing 10. Our algorithm determines two common blocks. $Block1$ contains all the triple patterns of the BGP except the last one and $Block2$ contains the last triple pattern of that BGP. $Block1$ is repeated in the OPTIONAL block. In Figure 3 (a), we see the operator graph. $Block2$ and $Block1$ are to be joined to get the bound values for the BGP. The bound values for $Block1$ are used to get values for both the BGP and the OPTIONAL block. The join output of $Block2$ and $Block1$ are then joined with $Block1$ again because the filter inside the OPTIONAL block needs to compare values bound for variables which come from both the BGP and the OPIONAL block. The second filter applied is the one outside the OPTIONAL block. After applying this filter operator, we get the query results.

We can handle any query with OPTIONAL blocks this way. A query might have nested OPTIONAL blocks and our approach can handle those queries too. We use Hadoop jobs to do the joins and apply the operators (e.g. Filter) in the query operator graph. However, the number of Hadoop jobs needed to do these operation is not necessarily equal to the number of operators. For the example operator graph shown in Figure 3 (a), we would need two jobs to do the joins and the filters. We can apply the filters in the same job which does the 2nd

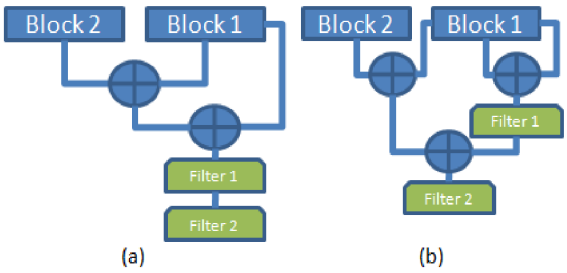

Fig. 3. Operator Graph for (a) SP^{2}B Query 6, (b) SP^{2}B Query 7

join. In general only join operators need separate Hadoop jobs. Other operators can be applied in the job doing the join which produces the output to be worked on by the operators.

An optional block might have multiple nested optional blocks inside it. Our approach can handle any number of such nesting in the optional blocks. Listing 11 shows SP^{2}B Query 7 which has a nested optional block inside the first optional block. Our algorithm identifies two unique blocks from the query. $Block1$ contains the third triple pattern of the BGP $?doc\ dc:title\ ?title$. $Block2$ contains the rest four triple patterns of the BGP. $Block2$ is repeated twice in the optional blocks. The operator graph we build for this query is shown in Figure 3 (b). We can see that $Block2$ is input three times, hence we can avoid reading from the disk thrice. Instead we only read data from the disk once for the block and reuse it three times. This significantly improves query running time.

Listing 11. SP^{2}B Query 7

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3)) }
```

Due to space constraint, we do not show the details of the trivial algorithm which builds the operator graph. These are commonly found in the relational database literature.

V. Results

In this section we present our performance with SP^{2}B dataset. Note that we have presented performance comparison between our framework and Jena, BigOWLIM and RDF-3X in [9]. We tested our approach on SP^{2}B dataset on a cluster of 10 nodes with POS schema. Each node had the same configuration: Pentium IV 2.80 GHz processor, 4 GB

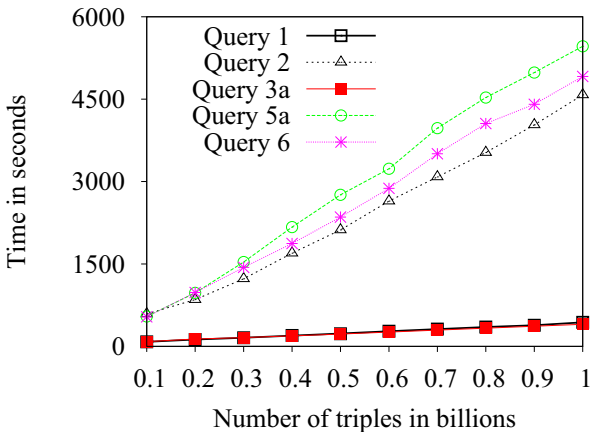

Fig. 4. Query Runtimes for SP^{2}B Queries

main memory and 640 GB disk space. We have selected some $SELECT$ queries which are representative of all the types of structures the query set has with $SELECT$ queries with $FILTERS$. Table III shows times to answer queries 1, 2, 3a, 5a and 6 by executing plans generated by *Relaxed-BestPlan* and *GeneratePlan* algorithms on different number of Triples. Queries 2 and 6 have $OPTIONAL$[15] blocks. We used *GeneratePlan* algorithm to generate plans for them. For the rest three queries, we used *Relaxed-Bestplan* algorithm to generate plans. The first column represents the number of triples in the range between 0.1 billion to 1 billion. Columns 2 to 6 of Table III represent the five selected queries from the SP^{2}B dataset [18]. Query answering time is in seconds. As expected, as the number of triples increased, the time to answer a query also increased. For example, Query 1 for 0.1 billion triples took 82.738 seconds whereas for 1 billion triples it took 436.817 seconds. Query 1 is simple and has only one joining variable, thus it took the least amount of time among all the queries. However, it has a three-way join. Query 2 also has ten triple patterns but only one joining variable. It has an $OPTIONAL$ block having only one triple pattern. Although it has a ten-way join, i.e. a star join between ten triple patterns, the performance of our framework is very good for this query. We can see this if we compare its running times with that of query 1. Query 1 has three-way join but the ten-way join of query 2 is much more complex. Compared to query 1, the running time of query 2 did not increase as much as the increase of complexity of the joins. For query 3a, we actually run the rewritten query shown in Listing 7. Our rewriting algorithm efficiently gets rid of all the complexities of the original query and to answer the query we don't have to do any join. We just read the input file, bind the project variable and output it. Query 5a has six triple patterns and one $FILTER$. Query 6 shows the efficiency of the plan generated by *GeneratePlan* algorithm. Even though query 6 has more triple patterns and filters than query 5a, because of

[15]http://www.w3.org/TR/rdf-sparql-query/#OptionalMatching

TABLE III
QUERY RUNTIMES FOR SP^{2}B QUERIES

Triples	SP^{2}B DATASET				
Billions	Q-1	Q-2	Q-3a	Q-5a	Q-6
0.1	82.74	590.87	85.16	543.34	537.35
0.2	124.38	851.02	133.43	973.31	980.27
0.3	156.62	1229.64	159.5	1534.79	1438.1
0.4	198.21	1697.17	188.82	2170.64	1872.5
0.5	229.25	2119.41	225.07	2757.87	2353.04
0.6	276.24	2643.51	260.7	3233.01	2876.72
0.7	314.32	3084.08	293.89	3972.62	3506.52
0.8	351.25	3528.28	331.64	4530.18	4056.44
0.9	384.70	4031.74	366.67	4984.35	4407.3
1	436.82	4578.66	402.61	5462.65	4914.38

identifying common blocks by the algorithm, its running times are comparable to that of query 5a.

Figure 4 shows the running times of the queries in a graph. The X axis represents number of triples in billions and the Y axis represents the time in seconds. As expected, the size of the dataset grows, the increase in time to answer a query does not grow proportionately. The increase in time is always less than the dataset size growth.

VI. CONCLUSIONS AND FUTURE WORKS

We have presented a distributed framework which can handle huge RDF graphs in a scalable and efficient manner. Our framework inherits scalability and fault tolerance from Hadoop, which it is based on. Our framework easily scales horizontally, because to increase capacity of our system all that needs to be done is to add new nodes to the Hadoop cluster. We have proposed a schema to store RDF data in plain text files, a query rewriting algorithm, a heuristics based greedy algorithm, *Relaxed-Bestplan*, to determine the best processing plan to answer a SPARQL query and the *GeneratePlan* algorithm to generate the best plan for queries having $OPTIONAL$ blocks. Our experiments demonstrate that our system is highly scalable. If we add data, the delay introduced to answer a query does not increase as much as the increment in the data size.

In the future, we would extend the work in few directions. First, we will run more experiments to compare our framework with other state-of-the-art frameworks for queries having $OPTIONAL$ blocks. Next, we will devise algorithm with other types of SPARQL queries e.g. ASK queries.

REFERENCES

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

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

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

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

[5] Cook, Stephen A., *The complexity of theorem-proving procedures*, Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, 1971, pp. 151–158.

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

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

[8] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu and Bhavani Thuraisingham, *Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools*, IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, pp. 1 - 10, Miami, FL.

[9] Mohammad Farhan Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani Thuraisingham, *Heuristics Based Query Processing for Large RDF Graphs Using Cloud Computing*, IEEE TKDE Special Issue on Cloud Data Management (to appear).

[10] Andrew W. Mcnabb, Christopher K. Monson and Kevin D. Seppi, *MRPSO: MapReduce particle swarm optimization*, GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation.

[11] Maged Michael, Jose E. Moreira, Doron Shiloach, Robert W. Wisniewski, *Scale-up x Scale-out: A Case Study using Nutch/Lucene*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.

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

[13] Thomas Neumann, Gerhard Weikum, *RDF-3X: a RISC-style engine for RDF*, Proc. VLDB Endow., Vol. 1, No. 1. (2008), pp. 647-659.

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

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

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

[17] Arnon Rosenthal and Upen S. Chakravarthy, *Anatomy of a Mudular Multiple Query Optimizer*, Proceedings of the 14th International Conference on Very Large Data Bases, VLDB 1988, pp. 230-239.

[18] Michael Schmidt, Thomas Hornung, Georg Lausen and Christoph Pinkel, *SP2Bench: A SPARQL Performance Benchmark*, 25th International Conference on Data Engineering (ICDE'09).

[19] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer and Dave Reynolds, , *SPARQL basic graph pattern optimization using selectivity estimation*, WWW '08: Proceeding of the 17th international conference on World Wide Web.

[20] Jeffrey Ullman, *Principles of Data Base Systems (2nd Ed)*, 1982.

[21] J. R. Ullmann, *An Algorithm for Subgraph Isomorphism*,J. ACM, vol. 23, issue 1, January, 1976, pp. 31–42, ACM, New York, NY, USA.

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

[23] Jesse Weaver and James A. Hendler, *Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples*, Proceedings of the 8th International Semantic Web Conference, 2009