ISSN: 2231-2307, Volume-1, Issue-ETIC2011, January 2012

Semantic Web Standard in Cloud Computing

Malini Siva, A. Poobalan

Abstract - **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. At the same time semantic web is also an emerging area to augment human reasoning. Resource Description Framework (RDF) which is semantic web technology that can be utilized to build efficient and scalable systems for Cloud Computing. A framework that can be built using Hadoop to store and retrieve large numbers of RDF triples by exploiting the cloud computing paradigm. To determine the Hadoop jobs, and 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. In the project, Hadoop's Map/Reduce framework is used to answer the queries. the results show that it can store large RDF graphs in Hadoop clusters built with cheap commodity class hardware.**

Index Terms – **Cloud Computing, Hadoop, Map/Reduce, RDF**

I. INTRODUCTION

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 (RDF) and SPARQL Protocol and RDF Query Language (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.

Malini Siva, Master of Computer Applications, Anna University, Chennai, India, 9443547203., (e-mail: sivalini@gmail.com).

A.Poobalan, Computer Science and Engineering, Anna University, Panruti, India, 9994463623

The main disadvantage with distributed systems is that they are optimized for relational data. They may not perform well for RDF data, especially because RDF data are sets of triples6 (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) can join on a single variable8 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.

Hadoop is a distributed file system where files can be saved with replication. In addition, it also contains an implementation of the Map/Reduce [6] programming model, a functional programming model which is suitable for the parallel processing of large amounts of data.

Our contributions are as follows:

1. We design a storage scheme to store RDF data in Hadoop distributed file system (HDFS10).
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.

The remainder of this paper is organized as follows: in Section II, we investigate related work. In Section III, we discuss our system architecture. In Section IV, we discuss how we answer an SPARQL query. In Section V, we present the results of our experiments. Finally, in Section VI, we draw some conclusions.

II. RELATED WORKS

Google uses Map/Reduce for web indexing, data storage, and social networking [5]. Yahoo! uses Map/Reduce extensively in its data analysis tasks]. IBM has successfully experimented with a scale-up scale-out search framework using Map/Reduce technology. In [3], researchers reported an interesting idea of combining Map/Reduce with existing relational database techniques. These works differ from our research in that we use Map/Reduce for semantic web technologies.

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. Jena [1] is a semantic web framework for Jena. 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. BigOWLIM [2] 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. RDF-3X [4] is considered the fastest existing semantic web repository. However, RDF-3X's performance degrades exponentially for unbound queries, and queries with even simple joins if the selectivity factor is low.

III. PROPOSED ARCHITECTURE

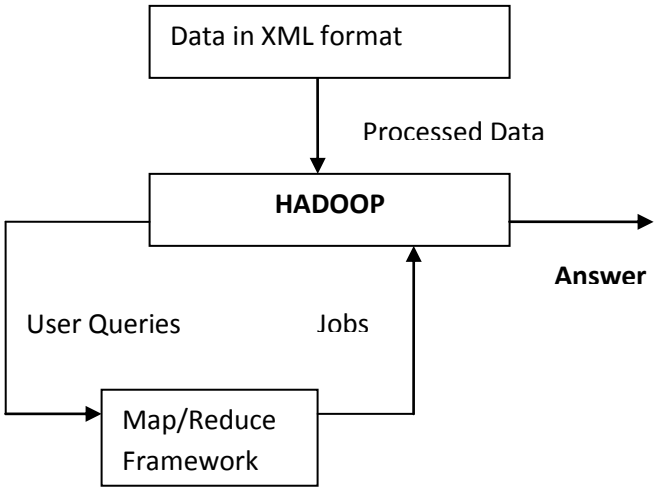


Fig. 1. The system architecture.

We have three subcomponents for data generation and preprocessing. We convert RDF/XML to N-Triples 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 B, C, and D.

Our Map/Reduce framework has three subcomponents in it. It takes the SPARQL query from the user and passes it to the Input Selector and Plan Generator. This component selects the input files, by using our algorithm, decides how many Map/Reduce jobs are needed, and passes the information to the Join Executer component which runs the jobs using Map/Reduce framework. It then relays the query answer from Hadoop to the user.

A. Data Generation and Storage

For our experiments, we use the LUBM [7] data set. It is a benchmark data set designed to enable researchers to evaluate a semantic web repository's performance . 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 Map/Reduce jobs. The processing steps to go through to get the data into our intended format are described in following sections.

B. File Organization

We do not store the data in a single file because, in Hadoop and Map/Reduce Framework, a file is the smallest unit of input to a Map/Reduce 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.

C. 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 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 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.

D. *Predicate Object Split*

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 c1; c2; . . . ; cn, then we will create files for each of these leaves and the file names will be like type_c1, type_c2; . . . , type_cn. Please note that the object values c1; c2; . . . ; cn are no 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.

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.

IV. MAP/REDUCE FRAME WORK

In this section, we discuss how we answer SPARQL queries in our Map/Reduce 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. The following section introduces the heuristics-based model we use in practice. Section C presents our heuristics-based greedy algorithm to generate a query plan which uses the cost model. We face tie situations in order to generate a plan in some cases and Section D talks about how we handle these special cases. Section D shows how we implement a join in a Hadoop Map/Reduce job by working through an example query.

A. 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.
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.
5. If the predicate is not rdf:type and the object is concrete, then we select all files for that predicate.

B. 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.

Heuristic Model

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 TaskTrackers19 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.

C. 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 practical model (Section 4.2.1) is computationally expensive. Then, we will present a heuristic and a greedy approach to generate an approximate solution to generate the best plan.

Computational Complexity of Reduced plan

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 Reduced plan 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 $e_i = (v_{i1}, v_{i2}) \in E$ represents a job that makes a transition from state vi1 to state v_{i2} . 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}.

Fig. 2. The (partial) graph for the running example query with the initial state and all states adjacent to it.

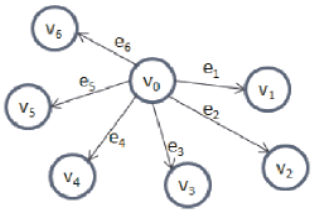
Running Example

```
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}
```

For example, in our running example query, the initial state v_0 = {X, Y, Z, XY, XZ},and the goal state, v_{goal} =Φ, 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 job1 = cost(job1), where cost is the given cost function. Fig. 2 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 Reduced plan, it can be shown that in the worst case, |V| - 2K, 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.

The worst case complexity of the Reduced plan problem is exponential in K, the number of joining variables in the given query.



v_0={X,Y,Z,XY,XZ}	
v_1={X,X,Z,XZ}	e_1=(v_0,v_1)=join(Y,XY)
v_2={X,Z,Z}	e_2=(v_0,v_2)=join(Y,XY), join(X,ZX)
v_3={X,Y,X,XY}	e_3=(v_0,v_3)=join(Z,XZ)
v_4={Y,Y,X}	e_4=(v_0,v_4)=join(Z,XZ),join(X,XY)
v_5={X,X,X}	e_5=(v_0,v_5)=join(Z,XZ),join(Y,XY)
v_6={YZ,Y,Z}	e_6=(v_0,v_6)=join(X,XY,XZ)

D. Reduced plan Problem and Approximate Solution

In the Reduced plan 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.

Algorithm 1. Reduce plan (Query Q)

```
1: Q← Remove_non-joining_variables(Q)
2: while Q≠Empty do
3:   J←1  // Total number of jobs
4:   U={u1,…,uk} ← All varibales sorted in
       //non-decreasing order of their E-counts
5:   Jobj←Empty // List of join operations in the
       // current job
6:  tmp←Empty // Temporaily stores resultant
      //triple patterns
7:  for i=1 to K do
8:    if Can-Eliminate(Q,ui)=true then
        //complete or partial elimination possible
9:      tmp←tmp ∪ Join-result (TP(Q,ui))
10:     Q←Q-TP(Q,ui)
11:     Jobj←Jobj ∪ join(TP(Q,ui))
12:   end if
13:  end for
14:  Q←Q ∪ tmp
15:  J ← J + 1
16: end while
17: return {Job1,….,Jobj-1}
```

Description of Algorithm 1. The algorithm starts by removing all the non-joining variables from the query Q. In our running example, Q = {X,Y,VZ,XY,XZ}, and removing the non-joining variable V makes Q ={X,Y, Z,XY,XZ}. In the while loop, the job plan is generated, starting from Job1. 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 E-count = 1, and X has E-count = 2. For each job, the list of join operations is stored in the variable Job_J , where Jis 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).

E. Map/Reduce Join Execution

In this section, we discuss how we implement the joins needed to answer SPARQL queries using Map/Reduce 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 Map/Reduce 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.

LUBM Query 2 shows the usage 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.

V. RESULTS

In this section, we present the benchmark data sets with which we experimented.

A. Data Sets

In our experiments with SPARQL query processing, we use two synthetic data sets: LUBM [7] and SP2B. The LUBM data set generates data about universities by using an ontology. 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.

VI. CONCLUSIONS

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.

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, "SWStore: A Vertically Partitioned DBMS for Semantic Web Data Management," VLDB J., vol. 18, no. 2, pp. 385-406, Apr. 2009.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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