IMPLEMENTATION AND OPTIMIZATION OF RDF QUERY USING HADOOP

YanWen Chen[1], Fabrice Huet[2] and YiXiang Chen[1]
[1]*Software Engineering Institute, East China Normal University, Shanghai, China*
[2]*INRIA, Sophia-Antipolis, Le Chesnay, France*

Keywords: Cloud Computing, MapReduce, RDF, Hadoop, Distributed Computing.

Abstract: With the prevalence of semantic web, a great deal of RDF data is created and has reached to tens of petabytes, which attracts people to pay more attention to processing data with high performance. In recent years, Hadoop, building on MapReduce framework, provides us a good way to process massive data in parallel. In this paper, we focus on using Hadoop to query RDF data from large data repositories. First, we proposed a prototype to process a SPARQL query. Then, we represented several ways to optimize our solution. Result shows that a better performance has been achieved, almost 70% improvement due to the optimization.

1 INTRODUCTION

Nowadays Resource Description Framework (RDF) is used to describe semantic data. At the same time, SPARQL query language has been developed for handling these data. Frameworks such as Jena and Sesame are provided to store and achieve semantic data, but these frameworks do not scale well for large RDF graphs. So distributed parallel processing, as a solution, has been explored to resolve the scalability problem for semantic web frameworks. Georgia, Dimitrios, and Theodore focused on designing SPARQL query mechanism for distributed RDF repositories (Georgia et al., 2008). Min, Martin, Baoshi and Robet paid more attention to scalable Peer-to-Peer RDF repositories (Min et al., 2004). Others did more work on querying RDF data with distributed computing technology. Hadoop which is mentioned in Tom's book (Tom, 2009), as a cloud computing tool, can process data in parallel, achieving a great improvement in performance. Also Hadoop provides a high fault tolerance because its distributed file system HDFS adopts a file replication mechanism. So now people start to focus on using Hadoop to process semantic data.

Mohammad, Pankil and Latifur proposed an architecture which includes file generator and query processor (Mohammad et al., 2009). In the file generator, RDF data file is split by predicates and then each file is divided further by objects. In the query processor, they proposed an algorithm to answer a SPARQL query. The main idea of the algorithm is to join several queries in one job. In the University of Queensland, scale-out RDF molecule store architecture (Andrew et al., 2008) has been proposed by Andrew, Jane and Yuan-Fang, in which instance data and ontology are converted to RDF in pre-processing phase. RDF graph is broken down into small sub-graphs which are then added into HBase (a Hadoop database). A query engine processes user's queries and returns results. Hyunsik, Jihoon, YongHyun, Min and Yon proposed a scalable query processing system named SPIDER (Hyunsik et al., 2009) which is also based on Hadoop. The idea of SPIDER is to store RDF data over multiple servers. Then sub queries are put to these cluster nodes. At last these sub query results are gathered and delivered to user.

In our work, we design a query engine which involves pre-processor and job trigger. The first module is responsible for dividing a SPARQL query into several sub queries according to its ontology. The second module contributes to creating jobs. Besides, three optimization tools (job reducing handler, query accelerator and file splitting handler) are devised to improve its performance.

The difference between our work and others are as follows: First, although we also propose a query engine to handle SPARAL query, our work focuses on handling ontology which is not considered by

Chen Y., Huet F. and Chen Y..
IMPLEMENTATION AND OPTIMIZATION OF RDF QUERY USING HADOOP.
DOI: 10.5220/0003387805120515
In *Proceedings of the 1st International Conference on Cloud Computing and Services Science* (CLOSER-2011), pages 512-515
ISBN: 978-989-8425-52-2

Mohammad in his paper (Mohammad et al., 2009). Second, we divide a complex query into sub queries and handle them using Hadoop while Andrew and his colleges focus on dividing RDF graph into small sub-graphs (Andrew et al., 2008). The most important point is we devise several optimizations tools to improve performance.

The rest sections are organized as follows. Section 2 describes our architecture. In section 3 we introduce our implementation. Section 4 details three optimizations. Section 5 presents some experiments and then reports their results. At last, section 6 reports the conclusions and future extensions.

2 ARCHITECTURE

We propose an architecture which includes one master node and several slave nodes (Figure 1).

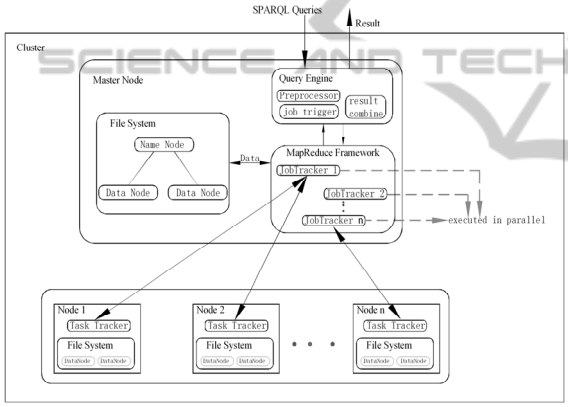

Figure 1: Proposed Architecture.

In master node, query engine consists of three modules: pre-processor module, job triggering module and result combining module. Pre-processor module is responsible for generating sub query groups. Job triggering module takes charge of triggering jobs (query and join). Result combining module is used to combine results of each group. After getting a job, master node assigns tasks to slave nodes in which tasks are executed in parallel.

Figure 2 shows us a query workflow with four stages: pre-processor stage, query stage, join stage, and result combining stage. Input is a SPARQL query. Output is a set of triples which meet the requirement of the SPARQL query. The pre-processor parses a SPARQL query into sub queries. After getting the results of these sub queries, we join them in the join stage. At last, the joining results from each group are combined into one file which is the final result of the SPARQL query.

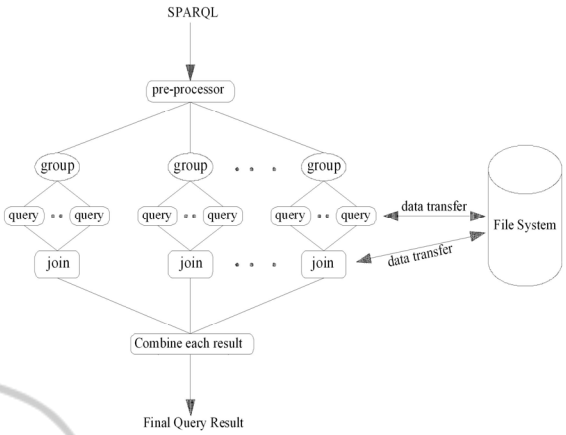

Figure 2: Workflow of a SPARQL query.

3 IMPLEMENTATION

A RDF file includes three columns: Subject, Predicate and Object. Before executing our query workflow, we store sorted RDF files in file system. Three sorted files which are sorted separately by Subject, Predicate and Object are generated from one original unsorted file. In our implementation, we firstly parse a SPARQL query in pre-process stage according to its Univ-Bench ontology. In the ontology there is a 'subClassOf' relationship among some classes. We create sub query jobs according to the relationship. For example, if 'Professor' has two subclasses 'FullProfessor' and 'AssociatedProfessor', then for a query like <?x type Professor>, three sub queries are generated--<?x type Professor>, <?x type FullProfessor> and <?x type AssociatedProfessor>. From each sub query, some parameters such as type, input file name, primary keyword, and secondary keyword can be gotten. At last, we use Hadoop to execute query job and then use Pig (a sub product of Hadoop) to execute join job.

4 OPTIMIZATION

4.1 File Splitting Handle

Before executing a map function, Hadoop splits input file into many small files and then creates Map-Reduce task for each split file. In our work we use File Splitting Handle to filter these split files. Since data has been sorted in each split file, the File Splitting Handle can select out files which may contain a certain keyword we need. So the performance will be improved since fewer tasks are

created. In our implementation, the first line record of each split file is compared with the certain word to determinate whether the file should be selected or not. If the first line record is larger than the certain keyword, then we discard the file, otherwise we choose it. The same strategy is used when reading a selected split file to avoiding a whole file searching.

4.2 Query Accelerator

Without using our accelerator, in each group, sub query jobs are executed sequentially. By using our accelerator, these jobs can be executed in parallel because we take advantage of multiple threads. Also in our accelerator, a join job is triggered as soon as at least two jobs are ready to be joined.

4.3 Job Reducing Handle

The job reducing handle is designed to reduce the number of jobs. In our implementation, when dividing SPARQL query into different groups, some identical sub queries maybe generated. So we use job reducing handle to detect these identical queries so that Hadoop can create only one job for them. Besides, in each group, if there is at least one query result is empty or an intermediate join result is empty, job reducing handle can find it and then stop creating jobs in this group, since the final result of the group must be empty.

5 EXPERIMENT AND RESULT

In our experiments, we investigate the performance of queries and the impact of optimizations from following aspects:

- Impact of job reducing handler optimization
- Impact of query accelerator optimization
- Impact of file splitting handler optimization

For these experiments we use a cluster with 14 nodes, each equipped with two dual core processors with 4GB of main memory and 72GB hard disk. The software we developed is written in Java 1.6 and based on Hadoop version 0.20.2. We use the LUBM benchmark (Yuanbo et al., 2005) designed by Yuanbo, Zhengxiang and Jeff to test query performance. In the benchmark, the number of RDF triples increases with increasing number of universities.

We use a simple query to test its query answering time varying sizes of datasets. Result shows it almost has a sublinear speed up.

5.1 Impact of Job Reducing Handler

We use the fourteen queries provided by the benchmark which includes 1000 university to investigate the relation between query answering time and the number of jobs. Result shows that query answering time has a linear speedup with the increasing number of jobs. Table 1 reports the impact of job reducing handler. Apparently the performance improved after using the handler. In order to know how much it is improved, we calculate the percentage of job decreasing and its performance improvement as shown in Table 2.

Table 1: Job numbers before and after job handling.

	Number of Jobs (before job reduction)			Number of Jobs (after job reduction)			Query Time(sec) (university1000)	
	Query	Join	Total	Query	Join	Total	before	after
Query 1	2	1	3	2	1	3	126	103
Query 2	12	9	21	6	8	14	882	480
Query 3	20	10	30	11	1	12	1260	240
Query 4	70	56	126	18	14	32	5292	900
Query 5	126	63	189	45	23	68	7938	1440
Query 6	4	0	4	4	0	4	161	109
Query 7	32	24	56	11	13	24	2352	900
Query 8	60	48	108	13	21	34	4536	1140
Query 9	480	400	880	28	53	81	36960	3120
Query 10	8	4	12	5	3	8	504	240
Query 11	4	2	6	4	2	6	140	120
Query 12	8	6	14	5	6	11	588	300
Query 13	168	84	252	46	24	70	10584	1440
Query 14	1	0	1	1	0	1	41	39

From the Table, we can see that, for query 2, 10, even they have the same percentage of total jobs decreasing, they still get a different improvement in performance. The reason is that their percentages of join jobs and query jobs are different. And according to our test, the more decreasing of join jobs, the better performance improvement we can get. So the query 10 has a better performance. Besides, from the table, we know that even there is no job decreasing like query 1, 6, 11 and 14, they still can get some performance improvement. It is because the impact of query accelerator. Some detail discussion will be given in the part 5.2.

Table 2: Percentage of performance improvement.

	query jobs decreased (%)	join jobs decreased (%)	total jobs decreased (%)	total time decreased (%)
Query 1	0.00%	0.00%	0.00%	18.25%
Query 2	50.00%	11.11%	33.33%	45.58%
Query 3	45.00%	90.00%	60.00%	80.95%
Query 4	74.29%	75.00%	74.60%	82.99%
Query 5	64.29%	63.49%	64.02%	81.86%
Query 6	0.00%	0.00%	0.00%	32.30%
Query 7	65.63%	45.83%	57.14%	61.73%
Query 8	78.33%	56.25%	68.52%	74.87%
Query 9	94.17%	86.75%	90.80%	91.56%
Query 10	37.50%	25.00%	33.33%	52.38%
Query 11	0.00%	0.00%	0.00%	14.29%
Query 12	37.50%	0.00%	21.43%	48.98%
Query 13	72.62%	71.43%	72.22%	86.39%
Query 14	0.00%	0.00%	0.00%	4.88%

5.2 Impact of Query Accelerator

In our query accelerator, multiple threads are used to make sub query jobs executed in parallel. After testing its impact, we found that performance has been improved by almost 30% in average. In addition, the numbers of jobs have directly influence on the performance improvement. So the four query mentioned in 5.1 have different improvement. Query 11 and query 6 which both have 4 sub query jobs should have a same improvement of performance and higher than query 1, 14. However the impact of join jobs cannot be ignored, which will reduce the performance. So the query 11 has a worse performance improvement than query 6 because the number of join jobs in query 11 is more than in query 6.

5.3 Impact of File Splitting Handler

We investigate the performance without and with file splitting handler using a dataset which includes 500 universities. In the experiments, we choose a certain data which are located at different positions of the dataset. Result shows that the closer to the end position the data is located, the lower performance we get, because more split files need to be read. The average improvement rate reaches to 74.53%.

6 CONCLUSIONS AND FUTURE EXTENSIONS

In this paper we address the problem of querying RDF data from large repositories and then investigate its performance. In order to improve the performance, several optimizations such as file splitting handler, query accelerator and job reducing handler have been explored.

As a conclusion, the work has returned encouraging results, almost 70% improvement according to the results of our experiments. Also due to these optimizations, the query time has a sublinear speedup with the increasing number of datasets. However, the performance of queries is not yet competitive because too many jobs are created, especially for the SPARQL query which has a wide hierarchy information or inference.

So in our future work, we will put more focus on the pre-processor module to reduce the number of jobs as much as possible. One interesting way could be to analyse each sub query before starting a job. The sub queries which read the same input file should to be combined as one job. However in this way, we should take care of the output and find a way to divide the output into different files.

REFERENCES

Andrew, N. and Jane, H. and Yuan-Fang, Li. (2008). *A Scale-Out RDF Molecule Store for Distributed Processing of Biomedical Data*. In Semantic Web for Health Care and Life Sciences Workshop, Beijing, China.

Georgia, D. S. and Dimitrios, A. K. and Theodore, S.P. (2008). *Semantics-Aware Querying of Web-Distributed RDF(S) Repositories*. In SIEDL2008, Proceedings of 1st Workshop on Semantic Interoperability in the European Digital Library, pp. 39-50, 2008.

Hyunsik, C. and Jihoon, S. and YongHyun, C. and Min, K. S. and Yon, D C. (2009). *SPIDER: A System for Scalable, Parallel/Distributed Evaluation of large-scale RDF Data*. In CIKM'09, November 2-6, 2009, Hong Kong, China, ACM 978-1-60558-512-3/09/11.

Min, C. and Martin, F. and Baoshi, Y. and Robet, M. (2004). *A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management*. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 2(2004) 109-130.

Mohammad, F. H. and Pankil, D. and Latifur, K. (2009). *Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce*. In CloudCom 2009, LNCS 5931, pp. 680-686. 2009.

Tom, W. (2009). *Hadoop: The Definitive Guide.* O'Reilly Media, Yahoo!Press.

Yuanbo, G. and Zhengxiang, P. and Jeff, H. (2005). *LUBM: A Benchmark for OWL Knowledge Base Systems*. Journal of Web Semantics, 3(2), pp.158-182.