Massive RDF Data Complicated Query Optimization Based on MapReduce

Jieru Cheng, Wenjun Wang, Rui Gao

School of Computer Science and Technology, Tianjin University, Tianjin, China

Abstract

Processing massive RDF data complicated query efficiently is a matter of concern for a long time in semantic web research. This paper proposes a MapReduce framework which could be a fast scalable solution. In this framework, the first step is doing the data preprocessing using a method called PredicateLead. Next the JobPartitioner algorithm partitions the query into several MapReduce jobs. The output of previous two operations should be the input of the last step—processing query in MapReduce. A case study on emergency decision demonstrates the efficiency and scalability of this framework.

© 2012 Published by Elsevier B.V. Selection and/or peer-review under responsibility of Garry Lee
Open access under CC BY-NC-ND license.
Keywords—massive RDF data; MapReduce; query partition

1. Introduction

Processing Massive RDF data complicated query slowly has been a bottleneck in the development of semantic web research. This problem is defined as follow:

Definition 1.1 Let P be the problem of massive RDF data complicated query.

\[ P = \{ F(s) \mid s \in A \land N(s) \geq 2 \land \exists t \in s \land B(t) = 2 \land C(t) \geq 10^9 \} \]

- A is the set of all SPQRQL query triples.
- F(s) is the whole query made up of several query triples.
- N(s) is the number of query triples.
- B(t) is the number of subject (or object) which is neither URI nor literal in one query (1 or 2).
- C(t) is the product of the sizes of RDF relations corresponding to the subject and object, which are appeared in one query and are neither URI nor literal.

The RDF query meeting the above description can only be called a "massive and complicated" problem. For this problem, many current query solutions are based on Jena uniprocessing optimized to achieve. But due to the uniprocessing limitation, these methods are not fast enough to implement immediate queries.
What’s more, their scalability is very poor. Here, "scalability" is defined as "to increase the amount of query data or query complication will not cause the augment of time complexity". If data are stored in database, regardless of how to optimize the index, the query will be subject to database processing power. So in order to break the above limitation, MapReduce [1] which is a cloud programming model raised by Google is introduced into the query processing.

The remainder of this paper is structured as follows: section 2 describes the related work about processing massive RDF data complicated query; section 3 provides the details of the three parts in the framework; section 4 will select an emergency decision query requirement to study and discuss the results of the case; finally, some conclusions and ideas for future work will be presented.

2. Related Work

There are several strategies on the storage of RDF data, the most common ones are: triples, the most popular storage solution in the early, such as Sesame [2], Jena [3], etc; vertical storage solution and the corresponding column storage database method [4] based on it; level storage solution. All these solutions provide ways to RDF storage and the improvement of query speed. They give us inspiration.

A lot of work will be equally committed to introduce cloud computing into RDF query research. Reference [5] advocates the combination of cloud computing and the improved method of Molecules[6] to improve the scalability of RDF query. But Molecules are not adapted to all requirements. Reference [7] promotes the use of an algorithm called DetermineJobs which also partitions a complicated SPARQL query into several MapReduce jobs. The disadvantage of this method is laying too much emphasis on efficiency, while ignoring the nature of the problem. One SPARQL query would like to find one specific property, but following the above method the last results can not the asked data. If an inverse approach is taken, it will also increase the cost of time, which may not be desirable.

Based on all above work, this paper proposes a framework which consists of three parts: data preprocessing, job partition and query execution. Massive RDF data complicated query can be processed in such a short time, what is more, just because the inherent scalability[8] of MapReduce, the processing time can be further reduced by simply increasing the number of computation nodes. The experiment uses the open source MapReduce framework Hadoop, and all input data will be stored in HDFS.

3. Algorithm description

In this section, descriptions of our framework for processing RDF query using MapReduce will be given in detail. As is shown in “Fig. 1”, the entire framework can be divided into 3 parts:

- Using PredicateLead method to preprocess the massive RDF data.
- Using JobPartitioner algorithm to partition the SPARQL query into individual jobs.
Executing these jobs for the query result. The query needs to be partitioned into individual jobs, based on the fact: Executing one job with MapReduce must satisfy the following premises; otherwise the correct result may not be got:

- Data should be independent.
- Different jobs should be independent.

The first condition can be guaranteed, for the minimal RDF data processing unit discussed here is a single line (one triple) in one file, and different lines are independent. In order to satisfy the second condition, partitioning the query into individual parts is required. Between the RDF triples appeared in a SPARQL query, there exists some reliance, which cannot be eliminated in one MapReduce job. So a solution is needed to partition the query into individual jobs, and each of them can be further divided into several independent subtasks. Each of these jobs can be executed using MapReduce.

A. Using PredicateLead Method to Preprocess the Massive RDF Data

Due to RDF data are massive, the size of these data files are quite huge. If storing these files in HDFS directly, for every query, it is necessary to scan all these files. This is very inefficient and will take large amount of time. For this problem, there are two solutions: first is to develop an efficient data structure to reduce the search complexity; second is to divide the data into different small parts for reducing the searching data amount. The second solution fits for our framework. The preprocessing using this solution will support the following MapReduce jobs well.

PredicateLead method will process RDF triples. For each triple, the method will generate a new pair in the form of <subject, predicate#object>, which corresponds to one single line in the final data file. Each line in the file will be used as a basic processing unit in MapReduce. The file name should be the corresponding predicate name. The entire processed files are stored in HDFS, which will be used as the input of jobs.

Table 1. JobPartitioner pseudo-code

<table>
<thead>
<tr>
<th>Algorithm1:JobPartitioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: QueryTriples q , Vertice targetVertice</td>
</tr>
<tr>
<td>Output: Jobs queue</td>
</tr>
</tbody>
</table>
B. Using JobPartitioner Algorithm to Partition the SPARQL Query into Individual Jobs

The algorithm is described using pseudo-code in “TABLE 1”.

For a given SPARQL query, a query graph is built: G= (V, E), V is the set of vertices, E is the set of edges. In a triple, subject and object act as vertices, and predicate act as the edge connected them. The input is query triples and the target vertice. Jobs are the final jobs queue, jobsKeyPoint records the “key” of each job, pointNum records the number of vertices in G by function getPointNum(), distanceQueue records the sorted distance sequence, connectionNum records the degree of each vertice.

The function updateConnectionNum() will update connectionNum according to the query graph G. The function getBFSDistance() will return the distance between each vertice and the target vertice using BFS, and function sortByDescending() will sort distanceQueue by distance in descending order. Traverse distanceQueue, for each vertice in distanceQueue, if the vertice’s degree is greater than or equal to 2, find all the query relations (i.e. the predicate relation represented by the edge adjacent to the vertice) related to it from the graph and generate a new job by function makeJobByPoint(). Current vertice’s value is the job’s key which will be recorded in jobsKeyPoint. After adding the job to the jobs queue, remove the related edge from the graph by function removeRelation(), and then update connectionNum. When all the vertices are visited, the jobs are the final job sequence.

```
1: jobs ← ∅; jobsKeyPoint ← ∅; distanceQueue ← ∅;
2: pointNum ← getPointNum(q); connectionNum ← ∅;
3: connectionNum ← updateConnectionNum(q);
4: distanceQueue ← sortByDescending
   (getBFSDistance(q, targetVertice));
5: for i=0 to pointNum do
6:       if connectionNum[distanceQueue[i]] ≥ 2 then
7:           Job
job ← makeJobByPoint(distanceQueue[i]);
8:           jobs ← jobs ∪ job;
9:         jobsKeyPoint ← jobsKeyPoint ∪ distanceQueue[i];
10:        q.removeRelation(distanceQueue[i]);
11:     end if
12: end for
13: return jobs;
```

Figure 2. SPARQL query.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel® Core™ 2 DUO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU E7500 @2.93GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>2GB</td>
</tr>
<tr>
<td>Network environment</td>
<td>Gigabit Ethernet</td>
</tr>
<tr>
<td>Operating system</td>
<td>Linux</td>
</tr>
<tr>
<td>Nodes number</td>
<td>32</td>
</tr>
<tr>
<td>MapReduce open source framework</td>
<td>Hadoop 0.20.2</td>
</tr>
</tbody>
</table>

Table 2 Experiment Environment
Figure 3: Multitasking query graph.

Figure 4: Contrast of query time.

C. Executing Jobs for the Query Results

The MapReduce jobs generated from JobPartitioner will be executed one by one, and the succeeding job may use the preceding jobs’ output as its input.

4. Experiment

This section will analyze a real requirement of emergency decision. We will show that, our method is efficient enough even for the query which needs to be processed in a very short time. There are 4 million RDF triples in the experiment, which represent all the political divisions of China and their coordinates (latitudes and longitudes). There are also 12 million RDF triples in the experiment, which represent the data of nearly all the enterprises in China, including their names, the districts they belong to, economic groupings, contact information, address etc. When a district has some disasters, it may need many kinds of supply from neighbor districts, such as: food, cloth etc. A decision may be made to contact the can food supplier around the district for food supply. The SPARQL query will be described in “Fig. 2”.

After partition the query into 2 jobs using JobPartitioner algorithm, the query graph is shown in “Fig. 3”. The algorithm can be further optimized by using some domain knowledge. For example: by classifying the enterprise data by their provinces, and according to different districts, we can estimate the information of its neighbor provinces, which will greatly reduce the number of RDF triples used for query.

Our experiment environment is shown in “Table 2”.

“Fig. 4” shows the results using three different query processing methods: uniprocessing (simple sequential method using sequential algorithm), multitasking (parallel methods without optimization using MapReduce), and area optimization (parallel methods with domain knowledge optimization using MapReduce). From the result, we can see that our framework using MapReduce have better performance. The processing time is less than 5 minutes, which is acceptable for real-time query of massive RDF data. The two results using MapReduce are much better than the result using sequential method, whose processing time exceeds 20 minutes or may be much more than that (we didn’t wait for the end). Besides, we can still see the improvement of domain knowledge. The results show that the performance has been improved by reducing the data amount using domain optimization. So it is recommended to collect as much domain knowledge from domain experts as possible for better optimization.

5. Conclusions And Future Work
This paper introduces a method for processing massive RDF data complicated query based on MapReduce. It shows that, by partitioning the query into individual jobs and executing each job with many different computation nodes, the processing time can be extremely reduced. Due to the inherent scalability of MapReduce, the processing time can be further reduced by increasing the number of computation nodes.

Independent MapReduce jobs can be parallelly processed. This is a focus of our further research.

Acknowledgment

We acknowledge the website mingluji (http://www.mingluji.com) for providing large amount of data for our experiments. This work is also supported by Science and Technology Support Key Project Plan of Tianjin (08zckfgx01600, 10ZCGYGX18600), Xinjiang Autonomous Region Science and Technology Support Project (200991111) and Science and Technology Research Project and New Product Trial Projects Plan of Guangxi (0993002-3).

References


