Web Semantics: Science, Services and Agents on the World Wide Web 10 (2012) 59–75

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: http://www.elsevier.com/locate/websem

WebPIE: A Web-scale Parallel Inference Engine using MapReduce

Jacopo Urbani *, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, Henri Bal

Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, Netherlands

ARTICLE INFO

Article history:
Available online 22 July 2011

Keywords:
Semantic Web
MapReduce
High performance
Distributed computing
Reasoning

ABSTRACT

The large amount of Semantic Web data and its fast growth pose a significant computational challenge in performing efficient and scalable reasoning. On a large scale, the resources of single machines are no longer sufficient and we are required to distribute the process to improve performance.

In this article, we propose a distributed technique to perform materialization under the RDFS and OWL *ter Horst* semantics using the MapReduce programming model. We will show that a straightforward implementation is not efficient and does not scale. Our technique addresses the challenge of distributed reasoning through a set of algorithms which, combined, significantly increase performance. We have implemented *WebPIE* (Web-scale Inference Engine) and we demonstrate its performance on a cluster of up to 64 nodes. We have evaluated our system using very large real-world datasets (Bio2RDF, LLD, LDSR) and the LUBM synthetic benchmark, scaling up to 100 billion triples. Results show that our implementation scales linearly and vastly outperforms current systems in terms of maximum data size and inference speed.

1. Introduction

Scalable reasoning is a crucial problem in the Semantic Web. At the beginning of 2009, the Semantic Web was estimated to contain 4.4 billion triples.[1] One year later, the size of the Web had tripled to 13 billion triples and the current trend indicates that this growth rate has not changed.

With such growth, reasoning on a Web scale becomes increasingly challenging, due to the large volume of data involved and to the complexity of the task. Most current reasoners are designed with a centralized architecture where the execution is carried out by a single machine. When the input size is on the order of billions of statements, the machine's hardware becomes the bottleneck. This is a limiting factor for performance and scalability.

A distributed approach to reasoning is potentially more scalable because its performance can be improved by adding more computational nodes. However, distributed reasoning is significantly more challenging because it requires developing protocols and algorithms to efficiently share both data and computation. The challenges concerning distributed reasoning can be grouped into three main classes:

- *Large data transfers:* Reasoning is a data intensive problem and if the data is spread across many nodes, the communication can easily saturate the network or the disk bandwidth. Therefore, data transfers should be minimized.
- *Load balancing.* Load balancing is a very common problem in distributed environments. In the Semantic Web, it is even worse because data has a high skew, with some statements and terms being used much more frequently than others. Therefore, the nodes in which popular information is stored have to work much harder, creating a performance bottleneck.
- *Reasoning complexity.* Reasoning can be performed using a logic that has a worst-case complexity which ranges from linear to exponential. The time it eventually takes to perform a reasoning task depends on both the considered logic and on the degree the input data exploits this logic. On a large scale, we need to find the best trade-off between logic complexity and performance, developing the best execution strategy for realistic datasets.

Reasoning is a task that can be performed either at query time (*backward reasoning*) or beforehand (*forward reasoning* or *materialization*). In some logics, it can be expressed as a set of *if-then* rules that must be applied until no further conclusions can be derived.

In this paper, we choose to follow a distributed approach to perform rule-based forward reasoning based on the MapReduce programming model.

The choice of MapReduce as programming model is motivated by the fact that MapReduce is designed to limit data exchange and alleviate load balancing problems by dynamically scheduling jobs

* Corresponding author.
E-mail addresses: jacopo@cs.vu.nl (J. Urbani), kot@cs.vu.nl (S. Kotoulas), jason@cs.vu.nl (J. Maassen), Frank.van.Harmelen@cs.vu.nl (F. Van Harmelen), bal@cs.vu.nl (H. Bal).

[1] http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics.

on the available nodes. However, simply encoding the rules using MapReduce is not enough in terms of performance, and research is necessary to come up with efficient distributed algorithms.

There are several rulesets that apply reasoning with different levels of complexity. First, we focus on the RDFS [14] semantics, which has a ruleset with relatively low complexity. We propose three optimizations to address a set of challenges: ordering the rules to avoid fixpoint iteration, distributing the schema to improve load balancing and grouping the input according to the possible output to avoid duplicate derivations.

Second, in order to find the best tradeoff between complexity and performance, we extend our technique to deal with the more complex rules of the OWL ter Horst fragment [18]. This fragment poses some additional challenges: performing joins between multiple instance triples and performing multiple joins per rule. We overcome these challenges by introducing three novel techniques to deal with a set of problematic rules, namely the ones concerning (`owl:sameAs`), (`owl:transitiveProperty`), (`owl:someValuesFrom`) and (`owl:allValuesFrom`).

To evaluate our methods, we have implemented a prototype called WebPIE (Web-scale Parallel Inference Engine) using the Hadoop framework. We have deployed WebPIE on a 64-node cluster as well as on the Amazon cloud infrastructure and we have performed experiments using both real-world and synthetic benchmark data. The obtained results show that our approach can scale to a very large size, outperforming all published approaches, both in terms of throughput and input size by at least an order of magnitude. It is the only approach that demonstrates complex Semantic Web reasoning for an input of 10^{11} triples.

This work is an extension of the work on RDFS reasoning published in [31], on OWL ter Horst reasoning published in [30] and our submission to the SCALE Challenge 2010 [29], which won the first prize. Compared to the previously published work, this paper contains a more detailed description of the algorithms, additional optimizations that further increase performance, support for incremental reasoning and a more thorough performance evaluation on more recent hardware.

This paper is organized as follows: first, in Section 2, we give a brief introduction to the MapReduce programming model. This introduction is necessary to provide the reader with basic knowledge to understand the rest of the paper.

Next, in Section 3, we focus on RDFS reasoning and we present a series of techniques to implement the RDFS ruleset using MapReduce. Next, in Section 4 we extend these technique to support the OWL ter Horst fragment. In Section 5 we explain how these algorithms can be further extended to handle incremental updates. In Section 6 we provide the evaluation of WebPIE. Finally, the related work and the conclusions are reported in Sections 7 and 8 respectively.

The techniques are explained at a high level without going into the details of our MapReduce implementation. In Appendix A, we describe the implementation of WebPIE at a lower level and we provide the pseudocode of the most relevant reasoning algorithms.

2. The MapReduce programming model

MapReduce is a framework for parallel and distributed processing of batch jobs [11]. Each job consists of two phases: a map and a reduce. The mapping phase partitions the input data by associating each element with a key. The reduce phase processes each partition independently. All data is processed as a set of key/value pairs: the map function processes a key/value pair and produces a set of new key/value pairs; the reduce merges all intermediate values with the same key and outputs a new set of key/value pairs.

2.1. A simple MapReduce example: term count

We illustrate the use of MapReduce through an example application that counts the occurrences of each term in a collection of triples. As shown in Algorithm 1, the *map* function partitions these triples based on each term. Thus, it emits intermediate key/value pairs, using the triple terms (s,p,o) as keys and blank, irrelevant, value. The framework will group all intermediate pairs with the same key, and invoke the *reduce* function with the corresponding list of values, summing the number of values into an aggregate term count (one value was emitted for each term occurrence).

Algorithm 1. Counting term occurrences in RDF NTriples files

```
map(key, value):
  // key: line number
  // value: triple
  emit(value.subject, blank); // emit a blank value, since
  emit(value.predicate, blank); // only number of terms
   matters
  emit(value.object, blank);
reduce(key, iterator values):
  // key: triple term (URI or literal)
  // values: list of irrelevant values for each term
  int count=0;
  for (value in values)
  count++; // count number of values, equaling occurrences
  emit(key, count);
```

This job could be executed as shown in Fig. 1. The input data is split in several blocks. Each computation node operates on one or more blocks, and performs the map function on that block. All intermediate values with the same key are sent to one node, where the reduce is applied.

2.2. Characteristics of MapReduce

This simple example illustrates some important elements of the MapReduce programming model:

- since the *map* operates on single pieces of data without dependencies, partitions can be created arbitrarily and can be scheduled in parallel across many nodes. In this example, the input triples can be split across nodes arbitrarily, since the computations on these triples (emitting the key/value pairs), are independent of each other;
- the *reduce* operates on an iterator of values because the set of values is typically far too large to fit in memory. This means that the reducer can only partially use correlations between these items while processing: it receives them as a stream instead of a set. In this example, operating on the stream is trivial, since the reducer simply increments the counter for each item;

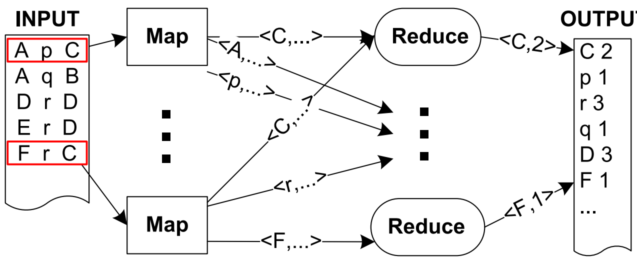

Fig. 1. MapReduce processing.