Towards Memory-Efficient Answering of Tree-Shaped SPARQL Queries Using GPUs

Jedrzej Potoniec[1] and Michal Madziar[2]

[1] Poznan University of Technology, Poland jpotoniec@cs.put.poznan.pl
[2] nSense Polska

Abstract. We present an idea of efficient query answering over an RDF dataset employing a consumer-grade graphic card for an efficient computation. We consider tree-shaped SPARQL queries and static datasets, to facilitate data mining over RDF graphs in warehouse-like setups.
Reasons to see the poster: a) presentation of the approach with examples; b) possibility of discussion about the implementation details.

1 Introduction

In this paper we consider answering a limited subset of SPARQL [9] SELECT queries over a static RDF dataset with the simple entailment [2]. This task is an important one from the perspective of semantic data mining systems [5]. We aim to answer the question if it is feasible to build an RDF store, that would efficiently use computer resources (esp. resources we tend to forget about, such as graphic cards) and that could be embedded into a data mining or decision support system and used directly on a workstation.

To define a class of considered queries, we reefer to a basic intuition how SPARQL Basic Graph Pattern (BGP) can be represented as a graph, with nodes corresponding to terms and variables and properties being arcs' labels. We consider queries with a single variable in the head, with WHERE clause consisting of a single BGP, whose graph is a tree, every node label is either an URI or a variable and every arc label is an URI. For short, we call such a query a tree-shaped query. Such queries are posed in high quantities to RDF stores by systems such as Fr-ONT-Qu [5].

Rest of the paper is organized as follows: in the Section 2 we present a brief overview of a similar research, in the Section 3 we describe details of our solution and we sketch future directions of our research in the Section 4.

2 Related work

Different aspects of processing semantic data on a graphics processing units (GPUs) have been considered over a past few years. In the work [6], the authors sketch a solution combining GPUs and CPUs to perform a fast aggregation and a basic reasoning over RDF streams. In [3] a problem of an efficient parallel

RDFS rule-based reasoning is tackled, especially with consideration for duplicated triples caused by multiple ways of inferring the same result. In [10] there are considered methods for RDF indexing based on B^+ trees and optimized for NVIDIA CUDA platform[3]. Authors of [11] analyzed parallel join algorithms, which are used to combine multiple SPARQL triple patterns into a solution for a query. [8] employs the *Rete Match* algorithm to perform parallel rule-based reasoning over RDFS knowledge bases.

There are numerous commercial-grade RDF stores available on the market, such as *GraphDB*[4], *Virtuoso*[5] or *Allegro Graph*[6], to just name a few. Yet, we are not aware of any such a store employing computational power of GPUs.

3 Approach

In this section we present a detailed description of our work. For the interested readers, the implementation is made available in the GitHub repository `https://github.com/jpotoniec/MagicStore`.

3.1 Loading phase

As data are loaded once and queried multiple times, we prefer to compute as much as possible during the loading. This phase consists of two steps: building dictionaries, when we apply Huffman coding [4] to URIs to fit as many triples as possible into limited amount of graphics card memory; and building the index, when the triples are sorted and packed into tight memory structures to allow for an efficient search.

In the step of building dictionaries, the number of occurrences of every URI in the data is counted. To obtain shorter codes, the URIs occurring as predicates are counted separately from those occurring as subjects or objects. The results are then used to build dictionaries of Huffman codes. As this step employs only sequential reading and counting, it does not require a lot of memory.

In the step of building the index, the data are read again, and now every triple is coded as a triple of Huffman codes. The triples are then packed into a three-level index structure, as depicted in the Figure 1. The upper two levels consist of consecutive cells, each containing a Huffman code and a pointer to the next level, and cells in the bottom level contains only Huffman codes. Because of the characteristics of the queries and the employed search algorithm, the first level contains predicates, the second objects and the third subjects.

As the codes vary in length, so do the cells in the index. In every cell, the first two bits of the first byte indicate length of the code in full bytes. Such an approach wastes small amount of memory, because the codes rarely occupy full bytes. We tested a scenario, where the pointer starts in the next bit after the

[3] `http://www.nvidia.com/object/cuda_home_new.html`
[4] `http://ontotext.com/products/ontotext-graphdb/`
[5] `http://virtuoso.openlinksw.com/`
[6] `http://franz.com/agraph/allegrograph/`

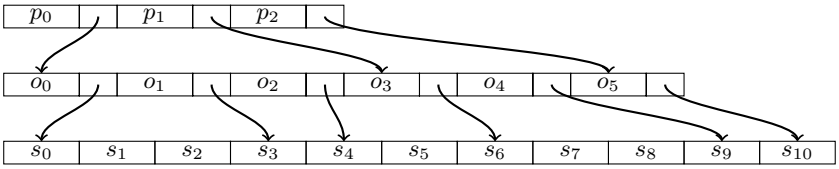

Fig. 1. Three level index employed by the system: the first level consists of predicates, the second level of objects and the third of subjects. Empty rectangles denote pointers to the next levels. There are 11 triples in this index: 6 with predicate p_0, 4 with predicate p_1 and 1 with predicate p_2, namely (s_{10}, p_2, o_5).

code, but it rendered out to be inefficient due to an additional overhead of bit shifting on GPUs.

To limit memory usage, triples are loaded in batches. During reading, triples are coded and stored in a vector. After the vector exceeds certain size, it is sorted, transformed into an index and merged with the index built so far.

3.2 Query answering phase

We employed a fairly simple, bottom-up query answering algorithm, based on iterating over the index built in the previous section. First, labels in a tree corresponding to a posed query are coded and then the Algorithm 1 is called for the root of the tree. Finally, returned values are decoded back to URIs.

Parallel for in the line 12 is realized as multiple calls of an OpenCL kernel [7], that looks for a particular predicate and object in the index and returns range of addresses in the third level of the index. This way only the first two levels of the index are required to be available to the GPU, what takes into account rather limited amount of memory available to the GPU. Merge in the line 14 is also implemented in parallel using merge part from the bitonic sort algorithm [1].

4 Conclusions and future work

In this paper, we presented a system for answering tree-shaped SPARQL queries over a static RDF dataset, a setup useful for semantic data mining. We believe that the presented system can be put to use as a local RDF store backing semantic data mining or decision support operations. We also think that this approach could be combined with a typical RDF store to provide a hybrid system, that uses GPU computations whenever possible.

In the future, we would like to compare it to normal RDF stores to ensure that this specialized solution really performs better. We also plan to extend supported SPARQL subset to literals and basic FILTER expressions (e.g. comparisons with a constant value).

Acknowledgement. Jedrzej Potoniec acknowledges support of Polish National Science Center, grant DEC-2013/11/N/ST6/03065.

```
1  Function Answer(n)
2      p ← a predicate labelling an arc leading to n
3      if n has no children then
4          if label of n is an URI then
5              return subjects occurring with a predicate p and an object labelling n
6          else
7              return subjects occurring with a predicate p and any object
8      else
9          O = ⋂_{c∈children of n} Answer(c)
10         if O = ∅ then return ∅
11         S = ∅
12         for o ∈ O do in parallel
13             S ← S ∪ {subjects occurring with a predicate p and an object o}
14         return merged sets from S
```

Algorithm 1: The basic query answering algorithm.

References

1. Cormen, T.H.: Introduction to algorithms. MIT press (2009)
2. Glimm, B., Ogbuji, C.: SPARQL 1.1 entailment regimes. W3C recommendation, W3C (Mar 2013)
3. Heino, N., Pan, J.Z.: RDFS Reasoning on Massively Parallel Hardware. In: Cudr-Mauroux, P., Heflin, J., et al. (eds.) The Semantic Web – ISWC 2012, Lecture Notes in Computer Science, vol. 7649, pp. 133–148. Springer Berlin Heidelberg
4. Huffman, D.: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE 40(9), 1098–1101 (Sept 1952)
5. Lawrynowicz, A., Potoniec, J.: Pattern based feature construction in semantic data mining. Int. J. Semantic Web Inf. Syst. 10(1), 27–65 (2014)
6. Makni, B.: Optimizing RDF Stores by Coupling General-purpose Graphics Processing Units and Central Processing Units. In: Aroyo, L., Noy, N.F. (eds.) Proc. of the Doctoral Consortium co-located with 12th Int. Semantic Web Conf. (ISWC 2013). CEUR Workshop Proceedings, vol. 1045, pp. 40–47 (2013)
7. Munshi, A.: The OpenCL Specification. Version 1.2. Tech. rep., The Khronos Group (Nov 2012)
8. Peters, M., Brink, C., et al.: Rule-based Reasoning on Massively Parallel Hardware. In: Liebig, T., Fokoue, A. (eds.) Proc. of the 9th Int. Workshop on Scalable Semantic Web Knowledge Base Systems, Sydney, Australia, October 21, 2013. CEUR Workshop Proceedings, vol. 1046, pp. 33–49. CEUR-WS.org (2013)
9. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C recommendation, W3C (Jan 2008)
10. Sankar, S., Singh, M., et al.: An Efficient and Scalable RDF Indexing Strategy based on B-Hashed-Bitmap Algorithm using CUDA. Int. J. of Computer Applications 104(7), 31–38 (October 2014)
11. Senn, J.: Parallel Join Processing on Graphics Processors for the Resource Description Framework. In: Beigl, M., Cazorla-Almeida, F.J. (eds.) ARCS '10 - 23th Int. Conf. on Architecture of Computing Systens 2010, Workshop Proceedings, February 22-23, 2010, Hannover, Germany. pp. 23–30. VDE Verlag (2010)