Large Scale Rule-Based Reasoning Using a Laptop

Martin Peters[1](✉), Sabine Sachweh[1], and Albert Zündorf[2]

[1] Department of Computer Science, University of Applied Sciences Dortmund, Dortmund, Germany
{martin.peters,sabine.sachweh}@fh-dortmund.de

[2] Software Engineering Research Group, Department of Computer Science and Electrical Engineering, University of Kassel, Kassel, Germany
zuendorf@cs.uni-kassel.de

Abstract. Although recent developments have shown that it is possible to reason over large RDF datasets with billions of triples in a scalable way, the reasoning process can still be a challenging task with respect to the growing amount of available semantic data. By now, reasoner implementations that are able to process large scale datasets usually use a MapReduce based implementation that runs on a cluster of computing nodes. In this paper we address this circumstance by identifying the resource consuming parts of a reasoner process and providing a solution for a more efficient implementation in terms of memory consumption. As a basis we use a rule-based reasoner concept from our previous work. In detail, we are going to introduce an approach for a memory efficient RETE algorithm implementation. Furthermore, we introduce a compressed triple-index structure that can be used to identify duplicate triples and only needs a few bytes to represent a triple. Based on these concepts we show that it is possible to apply all RDFS rules to more than 1 billion triples on a single laptop reaching a throughput, that is comparable or even higher than state of the art MapReduce based reasoner. Thus, we show that the resources needed for large scale lightweight reasoning can massively be reduced.

Keywords: Large scale reasoning · Rule-based reasoning · GPU · RETE algorithm · Memory efficient · Triple compression

1 Introduction

Semantic data and ontologies are used in a wide area of application like biomedical applications, smart environments and of course the Semantic Web. To be able to fully explore the existing data and for example to ensure a complete result set for queries, reasoners are used to derive facts that are implicitly given by the existing data. Thus, the reasoning process is one key feature when using semantic technologies. Nevertheless, with respect to the growing amount of data we face the challenge to provide a fast, scalable and efficient reasoning process.

F. Gandon et al. (Eds.): ESWC 2015, LNCS 9088, pp. 104–118, 2015.
DOI: 10.1007/978-3-319-18818-8_7

This problem was already addressed by different approaches, where most of them use a MapReduce based implementation to distribute the workload to a cluster of computing nodes [1–3].

While MapReduce based reasoners have turned out to be highly scaleable and efficient when using an adequate number of computing nodes, they are also complex and costly to deploy. On the other side, most of the real world datasets that are used in the scientific community have a size varying from a few million statements to up to a few billion statements. For example the Bio2RDF[1] portal provides different biomedical datasets with a size varying from less than 100 k statements to about 5 billion statements. One other often used semantic datasets is DBpedia [4], which is derived from Wikipedia and contains about 400 million statements in the English version. The real need resulting from these observations is to be able to process datasets with up to a few billion triples on a simple and affordable hardware like a well equipped laptop or a single workstation.

In our previous work [5,6] we introduced a rule-based reasoner that makes use of the massively parallel hardware of graphic cards (GPUs). The work is based on the RETE algorithm [7], which was introduced by Charles Forgy and is a widely used algorithm to implement production systems. Unlike most of the related work in the area of fast and scalable reasoning, which implements a static semantics (the semantics describes which implicit given facts shall be derived by the reasoner), the use of the RETE algorithm allows to define the semantics using simple rules that are provided by a rule-file and thus can easily be edited. In [6] we showed that our approach scales in a linear way for simple rulesets like RDFS on datasets with up to one billion triples on a single computing node. Nevertheless, the RETE algorithm and thus our implementation was quite memory consuming which is why we had to use a server with 192 GB of memory to be able to process one billion triples. This means that even a dataset with a few hundreds of millions of statements can easily exceed the capabilities of simple hardware like a laptop.

In this paper we address the aforementioned problems and introduce new concepts for an efficient reasoning process using the GPU on limited hardware[2] in terms of available memory. In detail we provide solutions to reduce the memory consumption of the RETE algorithm as well as of the data structures that are needed to efficiently identify duplicate triples. Furthermore, for an efficient execution, we introduce an approach that generates the source code executed on the GPU during runtime with respect to the given set of rules. After a short introduction of the RETE algorithm in Sect. 2 we start with an evaluation of the memory critical parts for a reasoner implementation. We are going to point out the aforementioned aspects in more detail and give examples on how much memory is actually needed to process different datasets. Based on these findings we introduce an adapted use of the RETE algorithm in Sect. 3, which allows to make heavy use of the hard disk instead of using the main memory. Section 4

[1] http://bio2rdf.org/.

[2] Limited hardware in this paper is understood as single computers like laptops or workstations.

finally addresses the need to hold all triples in main memory for a fast identification of duplicate triples that get inferred during the reasoning process. To reduce the memory consumption for deduplication, a memory efficient triple representation based on different approaches like differential encoding and variable byte coding is introduced.

2 Using RETE for a Reasoner Implementation

The RETE algorithm [7] is a pattern matching algorithm which can be used to implement production systems. Because the semantics that defines which implicit given facts should be materialized during the reasoning process can often be expressed in a rule-based way, like RDF Schema (RDFS) and pD* [8], the RETE algorithm can also be used to implement a reasoner. This not only results in a semantics independent implementation, but also allows to apply application specific rules.

2.1 Basic Concept

To introduce the RETE algorithm, we use two rules from the RDFS semantics that build the ruleset for an example:

$$(?x\ \ ?p\ \ ?y) \rightarrow (?p\ \ \text{rdf:type}\ \ \text{rdf:Property}) \tag{R1}$$

$$(?x\ \ ?p\ \ ?y)\ \ (?p\ \ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \tag{R2}$$

The first step of the algorithm is to build a RETE network. The network consists of different nodes $n \in N$, which can be alpha or beta nodes. Each unique rule term is mapped to one alpha node. A beta node in turn always has exactly two parent nodes (which can be alpha or beta) and may connect for example the two alpha nodes that are created from the two rule terms of R_2. The resulting RETE network from R_1 and R_2 is depicted in Fig. 1.

After the network was created, the matching process starts by applying the alpha matching. This means that all input triples are matched against all alpha nodes to check if a given triple matches the condition of the alpha node. For α_1 in Fig. 1 every triple will match, because the whole rule term consists of variables

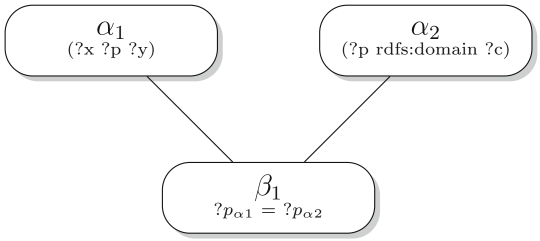

Fig. 1. RETE network of R_1 and R_2

(marked with a "?"). For α_2 a matching triple needs to have a predicate equal to "rdfs:domain". For every node in the network a working memory W is created that stores a reference to all matching triples. Based on the working memories of the alpha nodes, the beta matching can be performed. For β_1 this means that all matches that are stored in the working memory of α_1 are combined with all matches of α_2 to check if the combination of both is a match of β_1. That is the case, if the predicate of the α_1 match is equal to the subject of the α_2 match (or in other words if the elements of both matches marked with the variable $?p$ are equal).

Following up on the matching process, the rules can be fired to create new facts. For R_1 the working memory of α_1 is used as a fact basis because α_1 is the *terminal node* of R_1 and completely maps to the rule. Accordingly, for rule R_2, which consists of two rule terms, the working memory of β_1 is used to fire the rule. Using the new inferred facts the process starts again by iterating the new facts through the network (alpha and beta matching) and firing the rules, until no new triples are derived. For a more detailed description of how to use the RETE algorithm to infer new RDF facts we refer to [5] and [6].

2.2 Memory Consumption

After the short introduction of the RETE algorithm, next we are going to make some observations about the memory consumption of a RETE-based reasoner implementation. The following subsections introduce the major data structures that are necessary for such a reasoner implementation, while the last subsection gives a detailed overview about the particular memory consumption. We further assume that the triples are dictionary encoded, which means that each string is replaced by a numerical representation, where two identical strings are mapped to the same value. Thus, the dictionary encoding can be seen as a preprocessing of the input data which finally allows to operate on the data using more efficient numerical operations.

Triples. First of all the dictionary encoded triples need to be stored in memory for a reasoner implementation like described in [6]. The triples need to be stored in an array like data structure, where each triple can be addressed using a simple index, because this index is stored as a reference by the working memories of the RETE algorithm. Accordingly, using 8 byte datatypes for the numerical representation of one triple term, the memory footprint of the triple array is $n * 24$ byte, where n is the number of triples.

Working Memories. The second data structure is responsible to store all working memories W for the RETE algorithm. The size of the working memories highly depends on the used data- and ruleset. Furthermore, a single match in a working memory may take a single reference (the index of the referenced triple) like for alpha nodes, or multiple entries for beta nodes which always refer to multiple triples.

Table 1. Approximated memory consumption of triples and working memories for ρdf and RDFS assuming a load factor of 0.7 for the triple HashSet

Dataset	Ruleset	Total triples (n)	Triple size	Triple HashSet	Matches	References	Size of W	Total size
LUBM2000	ρdf	333.7 M	8009 MB	3814 MB	287.5 M	1022.5 M	8180 MB	20.0 GB
LUBM2000	RDFS	377.1 M	9050 MB	4310 MB	629.1 M	2119.1 M	16953 MB	30.3 GB
DBpedia	ρdf	400.6 M	9614 MB	4578 MB	123.1 M	446.7 M	3574 MB	17.8 GB
DBpedia	RDFS	475.1 M	11402 MB	5430 MB	554.1 M	1978.5 M	15828 MB	32.7 GB

Triple HashSet. Finally, a third data structure is needed that can be used to efficiently identify duplicate triples that may get inferred during the rule-firing of the reasoning process. These triples need to be rejected and should not be added to the triple list. As a minimal implementation this could be achieved by using a HashSet, where the value of the set stores the position of a triple in the triples-array. To check for a duplicate, the hash code of a new triple would be calculated to find the corresponding position in the HashSet and thus in the triple array. This would allow to check for a duplicate by one simple lookup in the HashSet and one more lookup in the triples-array (in case of hash collisions multiple lookups might be necessary). Because a HashSet should only be filled up to a specific load factor f (like 0.7) to reduce the number of collisions, an additional overhead of at least $(\frac{n}{f} - n)$ entries is necessary.

Total Memory Consumption. Table 1 gives a detailed overview of the size of the different data structures for different datasets that were processed with the ρdf [9] and RDFS ruleset. ρdf is a simplified version of the RDFS vocabulary and contains all RDFS rules with at least two rule terms. For the evaluation we used the Lehigh University Benchmark (LUBM) [10], which is an often used synthetic benchmark dataset that can easily be scaled to different sizes by defining the number of universities that shall be generated. To show the memory consumption, we generated 2000 universities which is why the dataset is called LUBM2000. Furthermore, we used DBpedia [4] (version 3.9) including all datasets of the English language as a real world dataset.

The size of the triples is directly calculated by the number of total triples (parsed and inferred). The number of matches and references are derived by an execution of the RETE algorithm using the given data- and ruleset. Note that the total memory consumption is only an approximation. Using Java, further overheads for example resulting from instantiating objects, may occur. Nevertheless, it can be seen that the total memory consumption for datasets with an input size of 270 M (LUBM2000) to 400 M (DBpedia) triples easily exceeds 17.8 to 32.7 GB, depending on the dataset and ruleset that was applied. In consequence, an adequate hardware like a workstation or server providing a large memory is necessary to be able to process datasets with the given size.

3 RETE on the GPU with an Adapted Working-Memory Concept

Modern GPUs provide a massively parallel hardware that may have much more computing power than modern CPUs if they are used in an appropriate way and are faced with a problem that can be highly parallelized. Thus, the challenge is to parallelize a problem in a way such that an optimal performance can be achieved when executed on a GPU.

The parallelization of the RETE algorithm for a rule-based reasoner implementation was already introduced in [5] and [6]. For alpha matching this means to create a thread on the GPU for every input triple (a thread or *work item* on the GPU is much more lightweight than on a CPU). Each thread is responsible for one triple and checks the match condition for every alpha node. If the triple does match an alpha node, it creates an entry in the corresponding working memory. During beta matching, all matches of one parent node ($m_{parent1} \in W_{parent1}$) need to be matched against all matches of the second parent node ($m_{parent2} \in W_{parent2}$). Therefore, a thread for every entry in $W_{parent1}$ is created that iterates through all matches in $W_{parent2}$ and checks if the combination of both matches meet the conditions of the beta node. This operation is performed for every beta node in the RETE network.

One disadvantage of the RETE algorithm is the high memory usage, which is caused by maintaining the working memories. Considering large datasets, a working memory can easily contain millions of entries (in the case of beta nodes the number of entries in a working memory can easily exceed the number of triples within the input dataset) that are references to the actual data. One way to reduce the amount of used memory would be to swap the working memories to the hard disk. Because the working memories are only accessed in bulks and no access to single entries is required, this would not cause much overhead in terms of load time. Nevertheless, this approach would still require to hold all triples in the main memory to be able to resolve the references contained in the working memories before the data can be processed. This is because for processing not only the references, but also the triples itself are needed.

Based on the previous considerations and in contrast to our previous work, an approach is needed that allows to fully swap the matches to the hard disk without the need to hold any additional data like the triples in main memory. To achieve this, we extend the use of the working memories to not hold a reference to the corresponding triples, but the matching elements of the triples itself. For α_2 from Fig. 1 for example the working memory would contain all values that correspond to the variables $?p$ and $?c$ from the matching triples. The working memory of β_1 in turn would hold four elements, which correspond to $?x, ?p, ?y$ and $?c$. Note that neither static elements of matches (like the rdfs:domain of α_2) nor double elements like the $?p$ in W_{β_1} are stored. A comparison of both approaches using working memories storing references and using working memories storing the actual data is depicted in Fig. 2.

While the resulting working memories will need more storage space than working memories storing only references, they can completely be swapped to

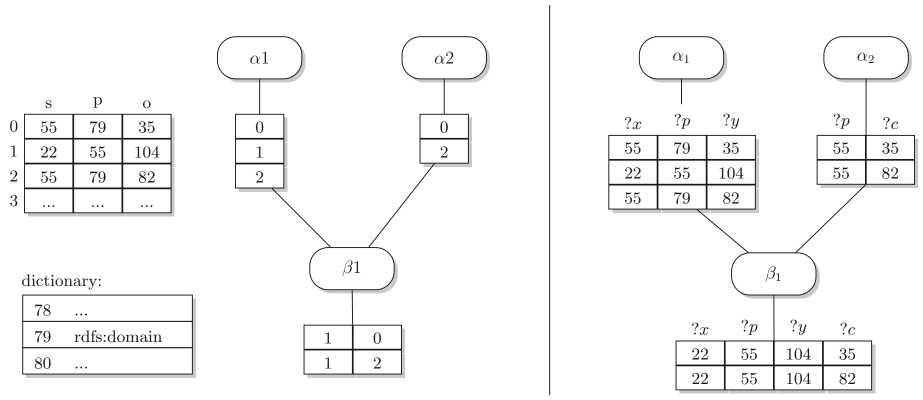

Fig. 2. RETE network with working memories using references (left) and RETE network with working memories using the full data excluding static data or recurring variables (right), for a dictionary encoded dataset

the hard disk. For a further processing, they can be read in blocks and directly be handed over to the corresponding task without the need to resolve any references. This allows to further minimize the usage of the main memory.

4 Compressed Triple-Index Structure

In the previous section we showed how the memory usage of the RETE algorithm can significantly be reduced by reorganizing the working memories and making use of the hard disk. One more problem that needs to be addressed is the memory consumption of the HashSet that is used in combination with the triples array to identify duplicate triples. To underline the importance of an efficient duplicate-lookup, Table 2 gives an overview of the number of triples that get inferred including the duplicates for the already known datasets. As can be seen, many of the derived triples are duplicates (up to 98.8 %). To be able to handle such an amount of lookups in an efficient way (more than 2 billion for DBpedia and RDFS), an in memory solution is necessary. To be able to reduce the memory

Table 2. Number of interred triples

Dataset	Ruleset	Derived triples	Unique triples	Duplicate triples	Percentage of duplicates
LUBM2000	ρdf	529.9 M	66.7 M	463.2 M	87.4 %
LUBM2000	RDFS	1813.6 M	110.1 M	1703.5 M	93.9 %
DBpedia	ρdf	580.1 M	7.0 M	573.1 M	98.8 %
DBpedia	RDFS	2263.6 M	81.5 M	2182.1 M	96.4 %

usage and to apply a compression to the triples needed for deduplication, one has to consider that each triple must only be compressed separately from other triples. This is necessary to be able to access the compressed version of each triple without decompressing other triples.

For triple compression we propose an index structure that stores the existing triples without loss of precision but with a much lower memory footprint. To do so, we make use of *vertical partitioning* [11], which is based on the fact that many datasets are only described by a few predicates. For each predicate in a dataset, a two column table is created that stores the subjects in the first column and the objects in the second column. Considering that the LUBM dataset only uses 32 predicates, this already results in a reduction of memory consumption of about 33 % (neglecting the overhead that is created by organizing the triples in 32 two-column tables instead of one three-column table).

For a further compression, we apply different well known techniques to the remaining two numerical values of a triple. First of all we try to reduce the amount of information to be encoded by checking if applying differential encoding could preserve memory. Differential encoding means for example to not store (s, o), but $(s, s - o)$. Making sure that $(s > o)$ and $((s - o) < o)$, the resulting value can be compressed more efficiently in the following steps. To avoid negative values resulting from differential encoding, we also have to make sure that the smaller value is subtracted from the bigger one. Finally, we map the subject as well as the object to a left value v_l and right value v_r in a way that the right value is always the smaller one, independently of the fact if differential encoding was applied or not. The calculation of v_l and v_r is described in Algorithm 1.

Algorithm 1. Calculation of v_l and v_r

```
Data: subject: $s$, object: $o$
Result: value left: $v_l$, value right: $v_r$
$v_l = s$;
$v_r = o$;
if $(s > o)$ then
    if $((s - o) < o)$ then
        $v_r$ = s - o;
else
    if $((o - s) < s)$ then
        $v_r$ = o - s;
        $v_l$ = o;
    else
        $v_l$ = o;
        $v_r$ = s;
```

The reason why we may reorder the subject and object in a way that the smaller value always becomes v_r is because we use variable byte encoding to store v_r. Variable byte encoding means to encode the data in units of bytes, where the lower-order seven bits are used to store the data and the eighth bit is used as an indicator of the end of a data value [12]: In particular the eighth bit is equal to 1 if the end of a data value is reached and 0 otherwise. The values $[0, 2^7)$ for example can be encoded using a single byte, where the first seven bits store the binary representation of the value and the eighth bit is equal to 1 to indicate the end of the value. Accordingly, the values in $[2^7, 2^{14})$ can be stored using two bytes, where the eighth bit of the first byte is equal to 0 and the eighth bit of the second byte is set to 1. The remaining 14 bits are used to encode the actual data.

Because we only store two numerical values in a row, we only have to encode v_r like described before. After reading a 1 at the eighth position of a byte, the start of the second value can explicitly be identified while the end of that value is defined by the remaining bytes. Thus, both values v_l and v_r can be encoded together in a single byte array that is explicitly sized to the amount of data that is needed to encode both values. To ensure that the encoded order of s and o is preserved as well as the fact if differential encoding was applied, two more bits are used, which are the most significant bits in the byte array.

For an example consider $s = 622$ and $o = 35$. Because $((s - o) > o)$, we do not apply differential encoding and get $v_l = s = 622$ and $v_r = o = 35$. The binary representation of 35 is 0010 0011, which results in 1010 0011 after applying variable byte encoding (the first bit was set to 1). The binary representation of 622 is 0000 0010 0110 1110, where the two most significant bits are used to point out if differential encoding was applied and if the order of the encoding of the subject as a left value and the object as a right value was preserved. Because we did not apply differential encoding, the most significant bit remains 0 and because we preserved the order of the s and o, the second most significant bit remains 0, too. Finally we can concatenate both binary values to

$$00\underbrace{00\ 0010\ \ 0110\ 1110}_{v_l}\ \ 1\underbrace{010\ 0011}_{v_r}$$

which can be encoded using three bytes. Assuming we are using 8 byte data types to dictionary encode the triple terms and we apply the aforementioned compression of vertical partitioning, differential encoding and variable byte encoding, we are able to reduce the amount of used data for the triple from the previous example from 24 byte to 3 byte without loss of precision. Nevertheless, the compression rate depends on the efficiency of vertical partitioning for a given dataset as well as on the respective value for s and o.

Based on the introduced triple-compression, a fast and memory efficient triple-index structure can be build to identify duplicate triples, which will be the only data structure that needs be be kept in memory during the reasoning process. For every predicate a HashSet can be created that stores the compressed value of s and o instead of storing for example two 8 byte values. This solution still allows to efficiently search for a duplicate triple by using hashing, but uses much less memory than the approach described in Sect. 2.2. Detailed information about the compression rate for different datasets is given in the next section.

5 Evaluation

In the following section we are going to evaluate the proposed concepts of an adapted working-memory for the RETE algorithm as well as the compressed index structure and give a detailed view on different aspects. Finally we are going to show how the concepts perform together in our reasoner implementation on different datasets.

5.1 Implementation

To evaluate our concepts, we used the reasoner implementation presented in [6] and adapted it to the new concepts and requirements. The reasoner is implemented in Java and uses OpenCL[3], the open standard for parallel programming of heterogeneous systems. It allows to program heterogeneous devices like GPUs and supports a wide range of parallelism. Furthermore, we use the jocl-library[4] as OpenCL Java bindings.

To allow an efficient execution of application code on the GPU, it is important to minimize memory access as well as to minimize the use of control flow structures like loops and if-then-else statements. To achieve these requirements, we also integrated a novel approach that generates the source code that is executed on the GPU during runtime. Based on the rule file that is given as an input when the reasoner execution is started, we generate the code that is executed on the GPU specific to the rules. This allows us to provide dedicated methods for example for each beta node that can be adapted to explicitly meet the needs of a single beta node and ensures to only load the data from memory that is needed. Furthermore, dictionary encoded values can directly be embedded to the code. This allows for example to check a triple to meet the conditions from α_2 in Fig. 1 (a triple predicate needs to be equal to "rdfs:domain") by directly compare a predicate to the numerical value of 79 instead of a variable (in Fig. 2 the dictionary encoded value for "rdfs:domain" is 79).

5.2 Datasets and Environment

Basically we use three different datasets. The Lehigh University Benchmark (LUBM) [10] was already mentioned before. It is widely used for reasoner evaluation and thus gives a good reference for a comparison. We generate datasets ranging from 1000 universities up to 8000 universities, which consist of more than 1 billion triples. In addition to the synthetic dataset, we also use the complete English version of DBpedia (version 3.9) as well as the real world Comparative Toxicogenomics Database (CTD) [13], which describes cross-species chemical-gene/protein interactions and chemical- and gene-disease relationships. All three datasets are used with the complete RDFS ruleset as it is defined by the W3C[5] as well as with the RDFS subset ρdf [9].

[3] http://www.khronos.org/opencl/.
[4] http://www.jocl.org/.
[5] http://www.w3.org/TR/2004/REC-rdf-mt-20040210/#RDFSRules.

Table 3. Reasoning results for ρdf and RDFS on a laptop

Dataset	Input triples	ρdf total triples	ρdf reasoning	ρdf throughput	RDFS total triples	RDFS reasoning	RDFS throughput
LUBM1000	134 M	167 M	41.6 s	4017 ktps	189 M	114.1 s	1653 ktps
LUBM2000	267 M	334 M	98.4 s	3391 ktps	377 M	287.6 s	1312 ktps
LUBM4000	534 M	668 M	296.9 s	2249 ktps	754 M	758.3 s	996 ktps
LUBM8000	1068 M	1335 M	716.7 s	1863 ktps	1509 M	1824.8 s	827 ktps
DBPedia	394 M	401 M	409.9 s	1154 ktps	475 M	2886.6 s	165 ktps
CTD	335 M	358 M	70.2 s	5104 ktps	358 M	306.8 s	1176 ktps

The evaluation is performed using an Apple MacBook Retina laptop from 2012 equipped with 16GB of memory, a 2.3 GHz Intel Core i7 processor, a 256GB SSD hard disk and a NVIDIA GeForce GT 650M graphic card with 1024MB of memory. All tests were performed five times and the average time of the whole reasoning process including materialization and deduplication is given.

5.3 Reasoning

The reasoning results in Table 3 show a decreasing throughput for both rulesets on the LUBM datasets. The highest throughput of 4017 kilo triples per second (ktps) is reached for LUBM1000 and the ρdf ruleset, which decreases to 1863 ktps for LUBM8000. The decrease is caused by different factors. First of all, we noticed that with a growing number of triples also the Java virtual machine garbage collector activity increases, which causes delays in the reasoner execution. Furthermore, the complexity of the beta-calculation of the RETE algorithm may grow in an exponential way for some rules, depending on the dataset. With respect to the used hardware, this further reduced the throughput.

Compared to ρdf, RDFS is not more complex, but causes more alpha nodes to be created during the RETE execution and materializes much more triples. This results in a more computation intensive execution and a lower throughput. Nevertheless, for RDFS we were able to reach a throughput ranging from 165 ktps (DBpedia) to 1653 ktps (LUBM1000). Furthermore, the full RDFS ruleset was successfully applied to more than 1 billion triples resulting in a total of 1.5 billion unique statements on a single laptop.

The memory consumption caused by the triple-index structure that is necessary to allow an efficient deduplication during the reasoning process is given in Table 4. It can be seen that the memory consumption per triple is between 5.94 and 7.45 byte, depending on the dataset. Thus, in comparison to a 24 byte triple representation we reached a compression of up to 75 %, which finally enables us to reason on large scale datasets like LUBM8000 on a hardware with only 16GB of memory. Because the triple information are kept in a hash structure, Table 4 gives also the used bytes per triple with overhead, which also considers the used memory for free entries in the HashSets. Depending on the reached density of values in the HashSets the overhead may differ in size.

Table 4. Memory usage of the triple-index structure after applying ρdf reasoning

Dataset	Predicates	Byte/triple	Byte/triple with overhead	Total memory
LUBM1000	32	6.04	9.71	1620 MB
LUBM2000	32	6.11	9.29	3098 MB
LUBM4000	32	5.69	8.26	5513 MB
LUBM8000	32	6.16	8.80	11748 MB
DBpedia	53139	7.45	12.81	5129 MB
CTD	43	5.94	9.36	3137 MB

Furthermore, Table 4 gives the number of predicates that are used within a dataset. As can be seen, for LUBM as well as for CTD the number of predicates is quite small such that the overhead when the vertical partitioning is applied is infinitesimal small. Even for DBpedia, where the number of predicates is much higher, the overhead that is caused by about 53k predicates is less than 8 MB for vertical partitioning, assuming that a single predicate causes an overhead of about 150 byte in our implementation.

Overall, the proposed concepts including the adapted RETE algorithm, which allows to completely swap the working memories to the hard disk, allow to reduce the main-memory consumption by more than 84 %. While using the naive approach for storing triple information we approximately used 20.0 GB of memory for applying ρdf Reasoning on LUBM2000. Using the new concepts, we only need about 3.1 GB for the compressed triple-index structure.

6 Related Work and Discussion

RDF compression has been investigated in the related work under several aspects. In [14] and [15] a binary representations for RDF graphs is used for a fast and memory efficient query answering. While query answering is not the purpose of the introduced triple-index structure, our goal was to efficiently identify duplicate triples, which can be done by checking the existence of a single (and unique) value. An OWL2 RL reasoner called RDFox that completely works in main memory is proposed in [16]. While the parallelization is applied similar to our work by creating multiple threads (one for each CPU core) that handle all triples one after the other, they report a memory consumption of at most 80 bytes per triple for creating the necessary index structures, which is about 10 times more than our implementation needs, but also serves a different purpose.

Large scale reasoning has recently been addressed in several works. While they may differ in the ontology language they implement, most of them have in common that they use a MapReduce implementation to handle the large amount of data and to be able to scale the architecture [1–3, 17]. In [17] the authors introduce WebPie, a MapReduce based implementation for RDFS and pD* reasoning.

They show that their architecture is highly scalable and is able to reason over 100 billion (LUBM) triples. On 64 computing nodes they reach a throughput of 481 ktps for 1 billion triples and a maximum throughput of 2125 ktps for 20 billion triples. The lower throughput for the smaller datasets is founded in the overhead that is introduced by the platform. While our implementation is not able to scale like WebPie, it is still able to reason over 1 billion triples and reaches a throughput of 1863 ktps on a laptop, which is nearly 4 times faster than WebPie for the same dataset. For smaller datasets our approach even reaches a throughput of 4017 ktps, which is almost twice as much as the maximum throughput reported in [17]. In [18] a parallel reasoner implementation is proposed that does not use MapReduce, but also distributes the workload to multiple computing nodes. The largest dataset used in [18] was a LUBM10000/4 (10.000 universities were generated, but only every fourth instance triple was used) with about 350M triples. The dataset was processed on 64 computing nodes each running four processes (each process was running on its own processor core). They reached a throughput of 1185 ktps (of input triples), but did not apply any deduplication.

A reasoner implementation that uses only a single computing node is described in [19]. The authors also use the massively parallel architecture of a GPU to apply the ρdf ruleset. Unlike our implementation, the work in [19] does not support user defined rules and is only able to process datasets that fit into the main memory of a single GPU. A high throughput for reasoning with the ρdf rules on a single machine is also reported for DynamiTE [20], which is a stream reasoner that was also evaluated by applying a full materialization. The reasoner makes use of multicore processors and reaches an input processing ratio of about 227 ktps for the LUBM8000 benchmark.

In comparison to our previous work [6] (using 192 GB memory and two GPUs reaching a maximum throughput of 2700 ktps for LUBM1000 and the ρdf rules), we were able to reduce the required hardware resources and increase the throughput at the same time. This was mainly possible by eliminating the need to resolve the references of working memories and by providing a faster implementation of the code that gets executed on the GPU by generating the code based on the given rules, leading to an overall increased throughput.

The presented concepts in this paper provide a holistic approach for large scale reasoning on limited hardware. Even though the hardware can be scaled in terms of using multiple GPUs, our approach does not allow to scale like a MapReduce based implementation mainly because the main memory is still a limiting factor. The throughput that is achieved depends on the structure of the dataset as well as on the ruleset and is particular high if the number of beta-matches that need to be computed is small. Thus, using more expressive and complex semantics, a MapReduce based approach can be more efficient due to the higher computation power. While we used only a single GPU from a laptop, the influence of these factors can be further reduced when multiple and more powerful GPUs are used, like described in [6]. Nevertheless, using lightweight ontology languages or appropriate user-defined rules, the proposed approach allows to reason on large datasets achieving a throughput that is comparable or even higher than state of the art reasoner reach on a cluster of computing nodes.

7 Conclusion

To the best of our knowledge, this work is the first one that shows a reasoner implementation that is able to apply the RDFS rules to a dataset with more than 1 billion triples using only a single laptop. This is possible by using a massively parallel execution in combination with a substantially reduction of the memory consumption of the whole reasoner process. To do so, we first introduced a concept to adapt the RETE algorithm by changing the way, working memories are used. Using the new concept, working memories can completely be stored to the hard disk without the need to hold all triples in memory. An efficient execution of the algorithm based on the new concepts was achieved by generating the source code that gets executed on the massively parallel hardware of a GPU based on the provided rules during runtime. This novel concept of applying a generative approach for the execution on parallel hardware allows to apply optimizations like reducing control flow structures and reducing memory access.

Furthermore, we introduced a compressed triple-index structure that allows to efficiently identify duplicate triples that get inferred during the reasoning process. The new triple-index structure has a memory footprint of about 25 % of the original dictionary encoded triple representation, which allows to keep much more triples for deduplication in memory. To achieve this, we used the vertical partitioning approach known from triple compression and combined it with different methods of integer compression and adapted them to our needs. Finally, we were able to process large scale datasets on a simple hardware without the need of a costly and time consuming setup of multiple computing nodes running in a cluster. While we did the evaluation using a laptop, a workstation equipped with more memory and a more powerful GPU (or even multiple GPUs) should be able to process even larger datasets.

After showing that GPUs are suitable to perform massively parallel reasoning on large datasets, our future work will include the investigation of adapting our approach to not only perform reasoning on static data, but also on data streams. Furthermore, an extension of the expressiveness that is supported by our rule-based reasoner will be part of the future work.

References

1. Urbani, J., Kotoulas, S., Massen, J., van Harmelen, F., Bal, H.: WebPIE: A web-scale parallel inference engine using MapReduce. Science, Services and Agents on the World Wide Web, Web Semantics (2012)
2. Liu, C., Qi, G., Wang, H., Yu, Y.: Reasoning with large scale ontologies in fuzzy pD* using MapReduce. Comput. Intell. Mag. **7**(2), 54–66 (2012)
3. Zhou, Z., Qi, G., Liu, C., Hitzler, P., Mutharaju, R.: Reasoning with Fuzzy-EL+ ontologies using MapReduce. In: Proceedings of the 20th European Conference on Artificial Intelligence, pp. 933–934. ECAI (2012)
4. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A Large-scale. Multilingual Knowledge Base Extracted from Wikipedia, Semantic Web Journal (2014)

5. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Rule-based reasoning on massively parallel hardware. In: 9th International Workshop on Scalable Semantic Web Knowledge Base Systems, pp. 33–49 (2013)
6. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: scaling parallel rule-based reasoning. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 270–285. Springer, Heidelberg (2014)
7. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In: Expert Systems, pp. 324–341 (1990)
8. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semant. Sci. Serv. Agents World Wide Web **3**(2–3), 79–115 (2005)
9. Muñoz, S., Pérez, J., Gutierrez, C.: Minimal deductive systems for RDF. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 53–67. Springer, Heidelberg (2007)
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semant. **3**(2–3), 158–182 (2005)
11. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 411–422. VLDB Endowment (2007)
12. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. In: Practice and Experience, Software (2013)
13. Mattingly, C.J., Rosenstein, M.C., Davis, A.P.P., Colby, G.T., Forrest, J.N., Boyer, J.L.: The comparative toxicogenomics database: a cross-species resource for building chemical-gene interaction networks. Toxicol. Sci.: Off. J Soc. Toxicol. **92**(2), 587–595 (2006)
14. Álvarez-García, S., Brisaboa, N.R., Fernández, J.D., Martínez-Prieto, M.A.: Compressed k2-triples for Full-In-Memory RDF engines. In: Association for Information Systems Conference (AMCIS) (2011)
15. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In: Proceedings of the 19th International Conference on World Wide Web, pp. 41–50. ACM (2010)
16. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel OWL 2 RL materialisation in centralised, Main-Memory RDF systems. In: Informal Proceedings of the 27th International Workshop on Description, pp. 311–323 (2014)
17. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part I. LNCS, vol. 6088, pp. 213–227. Springer, Heidelberg (2010)
18. Weaver, J., Hendler, J.A.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009)
19. Heino, N., Pan, J.Z.: RDFS reasoning on massively parallel hardware. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 133–148. Springer, Heidelberg (2012)
20. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: DynamiTE: parallel materialization of dynamic RDF data. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 657–672. Springer, Heidelberg (2013)