OWL reasoning with WebPIE: calculating the closure of 100 billion triples

Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri Bal

Department of Computer Science, Vrije Universiteit Amsterdam,
{j.urbani,kot,j.maassen,frank.van.harmelen,he.bal}@few.vu.nl

Abstract. In previous work we have shown that the MapReduce framework for distributed computation can be deployed for highly scalable inference over RDF graphs under the RDF Schema semantics. Unfortunately, several key optimizations that enabled the scalable RDFS inference do not generalize to the richer OWL semantics. In this paper we analyze these problems, and we propose solutions to overcome them. Our solutions allow distributed computation of the closure of an RDF graph under the OWL Horst semantics.
We demonstrate the WebPIE inference engine, built on top of the Hadoop platform and deployed on a compute cluster of 64 machines. We have evaluated our approach using some real-world datasets (UniProt and LDSR, about 0.9-1.5 billion triples) and a synthetic benchmark (LUBM, up to 100 billion triples). Results show that our implementation is scalable and vastly outperforms current systems when comparing supported language expressivity, maximum data size and inference speed.

1 Introduction

In this paper, we address the problem of *massively scalable OWL reasoning* and present WebPIE (Web-scale Parallel Inference Engine). In [15] we already presented a scalable and distributed method for materializing the closure of an RDF graph, using the RDFS semantics. That method encoded RDFS inference using the MapReduce framework, which allowed execution on a compute cluster.

In this paper, we extend our approach to deal with the complexity of the OWL semantics. We chose the OWL Horst fragment [8] of OWL because it provides a complete set of entailment rules represented as *if-then* rules. Because the rules in this fragment are more complex than the RDFS entailment rules, our previous approach is no longer sufficient. For example, previously, we could exploit the fact that all rules require a join between one schema triple and one instance triple. This observation underlies several key papers in this area [6, 15, 16], and allowed replicating schema triples in the main memory of all the nodes and performing the required joins on the fly. In the OWL Horst fragment, however, there are some rules that do not respect this pattern. As a result, this crucial optimization is no longer applicable.

Hence, the complexity of the OWL entailment rules required us to redesign our approach and to come up with novel optimizations that can deal with this higher complexity. In this paper we first recall the RDFS-specific optimizations (section 2), we then point out what are the major challenges for OWL reasoning and how our approach solves these problems (section 3). To evaluate our technique (section 4), we have implemented the WebPIE engine using Hadoop and performed experiments using both real-world and benchmark data. As real-world data, we have used the UniProt dataset[1], containing about 1.5 billion triples and the LDSR dataset[2] containing about 0.9 billion triples. As a benchmark, we have used the Lehigh University Benchmark (LUBM), for up to 100 billion triples. The obtained results show that our approach can scale to very large size, outperforming all published approaches, both in terms of triple throughput and maximum system size by at least an order of magnitude. To the best of our knowledge it is the only approach that demonstrates Semantic Web reasoning for an input in the order of 10^{11} triples.

2 Previous work: RDFS reasoning with MapReduce

To explain the use of MapReduce for reasoning, we first explain the basic idea of the framework, and then briefly recall the optimizations that we used to achieve efficient RDFS reasoning in our previous work, before turning to OWL Horst reasoning in section 3.

MapReduce is a programming model introduced by Google for large data processing [3]. The execution of a MapReduce program applies two user-specified functions, *map* and *reduce*, to the input data. The *map* function processes the input and outputs some intermediate key/value pairs. These pairs are partitioned according to the key and each partition is processed by a *reduce* function.

The closure of an RDF input graph can be computed by applying all rules iteratively on the input until no new data is derived (fixpoint). Single-antecedent rules can be easily implemented by iterating over the input and matching each triple individually. Applying rules is only challenging when there are multiple antecedents, since matching multiple antecedents means performing a join on the input triples, hence placing requirements on how to partition the data across compute nodes.

As an example, let us consider the rule from RDFS [5] which derives `rdf:type` based on the sub-class hierarchy:

$$s \texttt{ rdf:type } x,\ x \texttt{ rdfs:subClassOf } y \Rightarrow s \texttt{ rdf:type } y \qquad (1)$$

This rule effectively performs a join, which we can implement in MapReduce with a *map* and *reduce* function, as shown in Figure 1. In the *map* operation, we process each triple and output a key/value pair, using as value the original triple, and as key the triples term on which the join should be performed. In the case

[1] `http://www.uniprot.org`
[2] `http://www.ontotext.com/ldsr/`

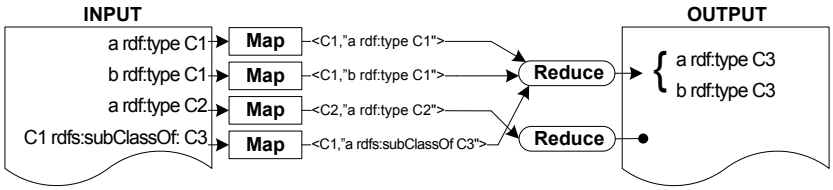


Fig. 1. Encoding the RDFS subclass-type rule in MapReduce.

of the above rule, to perform the sub-class join, triples with `rdf:type` should be grouped on their object ("x" in the rule's first antecedent), while triples with `rdfs:subClassOf` should be grouped on their subject (the "x" in the rule's second antecedent). When all emitted tuples are grouped for the reduce phase, these two will group on "x" and the reducer will be able to perform the required join. To calculate the complete closure, the application of all rules should be iterated, until fixpoint.

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

- since the *map* operates on single pieces of data without dependencies, input partitions can be created arbitrarily and can be scheduled in parallel across many nodes.
- 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.
- the *reduce* operates on all pieces of data that share a key. By assigning proper keys to data items during the *map* phase, the data is partitioned for the *reduce* phase. A skewed partitioning (i.e. skewed key distribution) will lead to imbalances in the load of the compute nodes. If term x is relatively popular, the node performing the *reduce* for x will be slower than others. To use MapReduce efficiently, we must find balanced partitions of the data.

A naive implementation of such RDFS reasoning is straightforward, but is inefficient because it produces duplicate triples (several rules generate the same conclusions), suffers from poor load-balancing and requires fixpoint iteration. In [15], we introduced three optimizations that vastly improved performance:

Loading schema triples in memory Typically, schema triples are far less numerous than instance triples. Furthermore, RDFS rules have at most one antecedent that is not a schema triple, so that no joins are required between instance triples. This allowed us to load the schema triples in the memory of each node and stream the instance triples, which improved load balancing.

Data preprocessing to avoid duplicates We have devised a way to partition triples in a manner that dramatically reduced duplicate derivations.

Ordering the application of the RDFS rules We have analyzed the RDFS ruleset and devised a rule ordering that removed the need to apply each rule more than once (no fixpoint iteration).

3 OWL Horst Reasoning with MapReduce

In this section, we will present an efficient implementation of OWL reasoning using MapReduce. First, we will define the logic we are interested in. Second, we will identify the main challenges it poses, in comparison with the optimizations presented in Section 2. Third, we will present efficient algorithms to address these challenges.

3.1 OWL Horst fragment

In this paper, we consider the Horst fragment of OWL [8]. The reasons for this choice are: (a) it is a *de facto* standard for scalable OWL reasoning, implemented by industrial strength triple stores such as OWLIM; (b) it can be expressed by a set of rules; and (c) it strikes a balance between the computationally unfeasible OWL full and the limited expressiveness of RDFS. The OWL Horst ruleset (formally known as pD) consists of the RDFS rules[5] (defined as D) and the rules shown in table 1 (defined as p). Our method performs forward inference. However, we should note:

- Similar to [15], we omit some rules with one antecedent (rules 5a,5b) as these can be parallelized efficiently and are commonly ignored by reasoners as yielding consequences that can also be easily simulated at query-time.
- We do not directly materialize inferences based on `owl:sameAs` triples. Instead, we construct a table of all sets of resources connected by `owl:sameAs` relationships (rules 6, 7, 9, 10 and 11 from Table 1). In other words, we represent the equivalence classes under `owl:sameAs`. Again, this is common practice in industrial strength triple stores. Note that this does not change the computational complexity of the task, since the `owl:sameAs` relationships are still calculated. The `sameAs`-table simply provides a more compact representation, reducing the amount of intermediate data that need to be processed and the size of the output.

3.2 Challenges in OWL reasoning

The OWL Horst rules in Table 1 show that the techniques presented in Section 2 are not always applicable. Here, we present some of the challenges for OWL reasoning.

No rule ordering For D, there is a rule execution order that allows us to compute the closure by executing each rule only once, in most cases. In the Horst fragment, there is no such ordering. Hence we must repeatedly apply rules until no new triples are derived (fixpoint iteration is required);

1:	p rdf:type owl:FunctionalProperty, u p v , u p w	$\Rightarrow$	v owl:sameAs w
2:	p rdf:type owl:InverseFunctionalProperty, v p u, w p u	$\Rightarrow$	v owl:sameAs w
3:	p rdf:type owl:SymmetricProperty, v p u	$\Rightarrow$	u p v
4:	p rdf:type owl:TransitiveProperty, u p w, w p v	$\Rightarrow$	u p v
5a:	u p v	$\Rightarrow$	u owl:sameAs u
5b:	u p v	$\Rightarrow$	v owl:sameAs v
6:	v owl:sameAs w	$\Rightarrow$	w owl:sameAs v
7:	v owl:sameAs w, w owl:sameAs u	$\Rightarrow$	v owl:sameAs u
8a:	p owl:inverseOf q, v p w	$\Rightarrow$	w q v
8b:	p owl:inverseOf q, v q w	$\Rightarrow$	w p v
9:	v rdf:type owl:Class, v owl:sameAs w	$\Rightarrow$	v rdfs:subClassOf w
10:	p rdf:type owl:Property, p owl:sameAs q	$\Rightarrow$	p rdfs:subPropertyOf q
11:	u p v, u owl:sameAs x, v owl:sameAs y	$\Rightarrow$	x p y
12a:	v owl:equivalentClass w	$\Rightarrow$	v rdfs:subClassOf w
12b:	v owl:equivalentClass w	$\Rightarrow$	w rdfs:subClassOf v
12c:	v rdfs:subClassOf w, w rdfs:subClassOf v	$\Rightarrow$	v rdfs:equivalentClass w
13a:	v owl:equivalentProperty w	$\Rightarrow$	v rdfs:subPropertyOf w
13b:	v owl:equivalentProperty w	$\Rightarrow$	w rdfs:subPropertyOf v
13c:	v rdfs:subPropertyOf w, w rdfs:subPropertyOf v	$\Rightarrow$	v rdfs:equivalentProperty w
14a:	v owl:hasValue w, v owl:onProperty p, u p v	$\Rightarrow$	u rdf:type v
14b:	v owl:hasValue w, v owl:onProperty p, u rdf:type v	$\Rightarrow$	u p v
15:	v owl:someValuesFrom w, v owl:onProperty p, u p x, x rdf:type w	$\Rightarrow$	u rdf:type v
16:	v owl:allValuesFrom u, v owl:onProperty p, w rdf:type v, w p x	$\Rightarrow$	x rdf:type u

Table 1. p ruleset.

Joins between multiple instance triples In D, at most one antecedent can be matched by instance triples. In p, rules 1, 2, 4, 7, 11, 15 and 16 contain two antecedents that can be matched by instance triples. Thus, loading one side of the join in memory (the schema triples) and processing instance triples in a streaming fashion no longer works because instance triples greatly outnumber schema triples and the main memory of a compute node is not large enough to load the instance triples.

Duplicate derivations The two challenges above contribute to a third challenge, namely generation of duplicates. First, since there is no rule ordering that can prevent fixpoint iteration, rules will be applied repeatedly and derive the same conclusions.

Multiple joins per rule In D, all the rules require at most one join between two antecedents. In p, rules 11, 15 and 16 require two joins.

In the next sections, we will describe how we overcome these challenges.

3.3 Overall structure

Our approach tackles the above challenges interleaving the application of the D rules and the p rules. Algorithm 1 summarizes the control flow of our algorithm.

Algorithm 1 RDFS/OWL reasoner: main control flow

```
calculate_closure(data):
  boolean first_time=true;
  while (true) {
    derived=apply_rules(data, D); // Apply D rules once
    if (derived == null && first_time == false)
      return data; // D derived nothing, return
     data= data + derived;

    do { // Do fixpoint iteration for p rules
      derived=apply_rules(data, p);
      data= data + derived; }
    while (derived != null);

    first_time=false; }
```

For D, we use the methods from [15], which will not be discussed. In this paper, we will deal with the p rules, for which fixpoint iteration is required.

On a rare occasion (when there are subproperties of *rdfs:subPropertyOf*), the implementation in [15] does not produce the full closure for D without fixpoint iteration. To prevent this, the overall algorithm will stop when the application of the D rules does not produce any new triple so that no derivation is left out.

If we take a closer look at the rules of the p fragment, we notice that some rules can be implemented exploiting the optimizations introduced for the RDFS reasoning. These are the rules *3,8a,8b,12a,12b,12c,13a,13b,13c,14a,14b*.

Furthermore, rules *1* and 2 require a join on subject and predicate or predicate and object for two instance triples and a join on predicate with a schema triple. We found that these rules were straightforward to implement by partitioning on subject and predicate or predicate and object while the others can be efficiently implemented exploiting the RDFS optimizations. Thus, we will exclude these rules from further discussion.

All other rules from p are indeed challenging and require detailed explanation: Section 3.4 deals with rule 4, Section 3.5 deals with rules 6, 7, 9, 10, 11 and Section 3.6 deals with rules 15 and 16.

3.4 Transitivity algorithm

Rule 4 of the Horst fragment requires a three-way join between one schema triple and two instance triples. It seems similar to rules 1 and 2, suggesting that it can be implemented by partitioning triples according to `(pw)` (i.e. partition triples according to subject-predicate and predicate-object) and performing the join in-memory, together with the schema triple. Nevertheless, there is a critical difference with rules 1 and 2: the descendant is likely to be used as an antecedent (i.e. we have chains of resources connected through a transitive relationship). Thus, this rule must be applied iteratively.

Applying rule 4 as above will lead to a large number of duplicates, because every time the rule is applied, the same relationships will be inferred. For a

Algorithm 2 owl:transitivity closure (p rule 4)

```
map(key, triple, n):
 //key: distance of the triple
 //triple: triple in input
 //n: current step

 if (key.step = 2^(n - 2) || key.step = 2^(n -1)) then
  emit({triple.predicate, triple.object}, {flag=L, key.step, triple.subject});
 if (key.step > 2^(n-2) then
  emit({triple.predicate, triple.subject}, {flag=R, key.step, triple.object});

reduce(key, iterator values):
  for(value in values) do
    if (value.flag = 'L')
      leftSide.add({key.step, value.subject})
    else
      rightSide.add({key.step, value.object})
  for(leftElement in leftSide)
    for(rightElement in rightSide)
      newKey.step = leftElement.step + rightElement.step //distance new triple
      emit(newKey,triple(leftElement.subject, key.predicate, rightElement.object));
```

transitive property chain of length n, a naive implementation will generate $O(n^3)$ copies while the maximum output only contains $O(n^2)$ unique pairs.

We can solve this problem if we constrain how triples are allowed to be combined. At the nth iteration of the algorithm we would like not to derive triples which have a graph distance less or equal than 2^{n-2} because these were already derived in the previous execution. We also would like to derive the new triples only once and not by different combinations. The conditions to assure this are:

- on the left side of the join (triples which have the key as object) we allow only triples with distance 2^{n-1} and distance 2^{n-2};
- on the right side of the join (triples which have the key as subject) we allow only triples with the distance greater than 2^{n-2}.

The complete picture is shown in Algorithm 2. The map function filters out all the triples that do not have a transitive predicate by checking the input with the in-memory schema and it selects the triples which suit the possible join by checking their distance value. The reduce function simply loads the two sets in memory and returns new triples with distances corresponding to the sums of the combinations of distances in the input. In the ideal case, this algorithm completely avoids duplicates. Nevertheless, when there are different chains that intersect, it will produce duplicate derivations, but much fewer than without this optimization.

3.5 SameAs algorithm

Rules 7 and 11 from p are problematic because they involve a two-way and a three-way join between instance triples. Similarly to before, since the join is on

the instance data, we cannot load one side in memory but instead we are obliged to perform the join by partitioning the input based on the part of the antecedents involved in the join.

However, in this case, this approach would cause severe load balancing problems. For example, rule 11 involves joins on the subject or the object of an antecedent with no bound variables. As a result, the data will need to be partitioned on subject or object, which follow a very uneven distribution. This will obviously lead to serious load balancing issues, since a single machine will be called to process a large portion of the triples in the system (e.g. consider the number of triples with `foaf:person` as object).

To avoid these issues, we first apply the logic of rule 7 to find all the groups of synonyms (e.g. resources connected by the `owl:sameAs` relation) that are present in the datasets and we assign a unique key to each of these groups. In other words, we calculate all non-singleton equivalence classes under `owl:sameAs`. We store the pairs `(resource, group_key)` in a table that we call the `sameAs`-table. Subsequently, we replace in our input all the occurrences of the resources in the `sameAs`-table with their corresponding group key. In other words, we use a single canonical representation for each equivalence class.

This procedure, which is common practice in existing reasoners, does not explicitly materialize all the derivations by rule 11, but instead produces a more compact representation of these results. This procedure brings as additional advantages that it reduces the complexity of a three-way join of rule 11 to a two-way join applied during the operation of replacement; and it makes the execution of the p rules 6, 9 and 10 trivial and redundant to implement.

We will now describe building the `sameAs`-table and replacing all items by their canonical representation.

Building the sameAs-table. Our purpose is to calculate all the groups of resources that are connected with the `owl:sameAs` relation. The graph with `owl:sameAs` relationships is undirected, since `owl:sameAs` is symmetric. We turn this into a directed acyclical graph by assigning all nodes id, and having edges point to the node with higher id. The node with the lowest id in a group will be the canonical representation for all nodes that are reachable from it. We now want to efficiently calculate all nodes reachable from the canonical node.

To this end, we have developed a MapReduce algorithm. The intuition behind this algorithm is that edges that create a shorter path to the canonical representation should be created incrementally and in parallel and edges that are no longer needed should be removed. Eventually, all edges will originate from the canonical representation. Algorithm 3 shows this process: the graph is partitioned across nodes and the outgoing edges of a node are replaced by the incoming edge from the node with the lowest id, if such a node exists. This process is repeated until no edges can be replaced.

Replacing resources with their canonical representation. Since our purpose is to replace in the original dataset the resources in the `sameAs`-table with

Algorithm 3 owl:sameas reasoning (p rule 7)

```
map(key, edge): //key: irrelevant
   //Partition graph across nodes
   emit(edge.from, {forward, edge.to});
   emit(edge.to, {backward, edge.from});

reduce(key, values):
   //key:
   // value: the nodes of the edge
   toNodes.empty(); // edges to other nodes
   fromNodes.empty(); // edges from other nodes
   fromNodes.add(key);
   for (value in values) // collect all incoming and outgoing edges to node
     if (value.forward) toNodes.add(value);
     else if (value.backward) fromNodes.add(value);
   for (to in toNodes)
     emit(null, {fromNodes.minValue(), to});
```

their canonical representation, we must perform a join between the input data and the information contained in the table. In principle, the join is executed by partitioning the dataset on the single term. Since the term distribution is very uneven, we suffer from a severe load balancing problem. We circumvent this problem by sampling the dataset to discover the most popular terms, and loading their eventual replacements in the memory of all nodes. (In our implementation, we typically sample a random subset of 7% of the dataset). When the nodes read the data in the input, they check whether the resource is already cached in memory. If it is, the nodes replace it on-the-fly and send the outcome to a random reduce task flagging it to be output immediately. For non-popular terms, the standard partitioning technique is used to perform the join, but since these terms are not popular, the partitioning will not cause load balancing issues.

Note that this approach is applicable to datasets with any popularity distribution: If we have a large proportion of terms that are significantly more popular than the rest, they will be spread to a large number of nodes, dissipating the load balancing issue. If there is a small proportion of popular terms, there will be enough memory to store the mappings.

3.6 someValuesFrom and allValuesFrom algorithm

Rules 15 and 16 present the following challenges: (a) they contain two joins, for v and for w. (b) one join is between antecedents that match many triples (`(`u p x`)` matches all triples, `(`x `rdf:type` w`)` matches a large subset). In this section, we will focus on rule 15. The algorithm for rule 16 is entirely analogous.

As with all schema triples, triples of the form `(`v `owl:someValuesFrom` w`)` and `(`v `owl:onProperty` p`)` are few and can be loaded into memory. The join between the schema triples can be done in the nodes' memory using the standard techniques. The join between the instance triples `(`u p x`)` and `(`x `rdf:type` w`)` is the most challenging part: since these triples are too many to fit in the memory of a single machine, they will need to be partitioned.

Algorithm 4 owl:someValuesFrom reasoning (p rule 15)

```
map(key, triple): //key: irrelevant
   joinSchema = join on the subject between someValuesFrom and onProperties triples
   if (triple.predicate == "rdf:type")
     if (triple.object in joinSchema.someValuesFromObjects)
          entries = joinSchema.getJoinEntries(triple.object)
          for (entry in entries)
            emit({entry.p,triple.subject}, {type=typetriple, resource=entry.
                 onPropertySubject});
   else if (triple.predicate in joinSchema.onPropertiesSet)
     emit({triple.predicate,triple.object}, {type=generictriple, resource=triple.subject});

reduce(key, iterator values):
   // key: partition (p,x)
   // values: parts of the instance triples used for the derivation
   types.clear(); generic.clear();
   for (value in values)
     if (value.type = typetriple) types.add(value.resource)
     else generic.add(value.resource)
   for (v in types)
     for (u in generic)
       emit(null, triple(u, "rdf:type", v));
```

To overcome memory limitations, these partitions should be small. Our initial implementation has filtered the instance triples which match against the two schema triples. Then, it partitioned them according to x. Since x is a single resource, one partition may contain many elements. Thus, the available memory of each node was not always sufficient to store the partition and perform the join in-memory. Consequently, this method was abandoned.

To reduce the size of the partitions, we have developed Algorithm 4. It aims at reducing the size of the partitions by performing the joins with the schema triples as soon as possible. We first perform the join between the two schema triples (`(v owl:someValuesFrom w)` $\bowtie$ `(v owl:onProperty p)`). Then, *before we partition*, we perform the join between the above and either `(u p x)` or `(x rdf:type w)`, calculating `(v owl:someValuesFrom w)` $\bowtie$ `(v owl:onProperty p)` $\bowtie$ `(u p x)` and `(x rdf:type w)` $\bowtie$ `(v owl:someValuesFrom w)` $\bowtie$ `(v owl:onProperty p)`. Now, in both cases, we have all possible bindings for x and p. Thus, we can partition on `(xp)` and perform the join during the reduce phase. Since partitioning is now done on two variables, each partition is much smaller. The pseudo code is shown in Algorithm 4.

4 Evaluation

We have used the Hadoop framework[3], an open-source Java implementation of MapReduce, to implement and test the algorithms explained in the previous sections[4]. Hadoop uses a distributed file system that uses the local disks of the

[3] `http://hadoop.apache.org`
[4] Our open source code is available at `https://launchpad.net/reasoning-hadoop`

participating machines, and manages execution details such as data transfer, job scheduling, and error management.

Our implementation was validated against the OWLIM reasoner[5]. It should be noted that our results do not completely coincide with those of OWLIM, since the latter offers limited support of owl:intersectionOf and owl:unionOf, which is not part of the Horst semantics.

We have performed the experiments on the Vrije Universiteit cluster of the DAS-3 distributed supercomputer[6] using up to 64 compute nodes. Nodes were equipped with two dual-core 2.4GHz Opteron processors, 4GB of main memory, 250GB hard disk and a Gigabit Ethernet interconnect.

4.1 Datasets

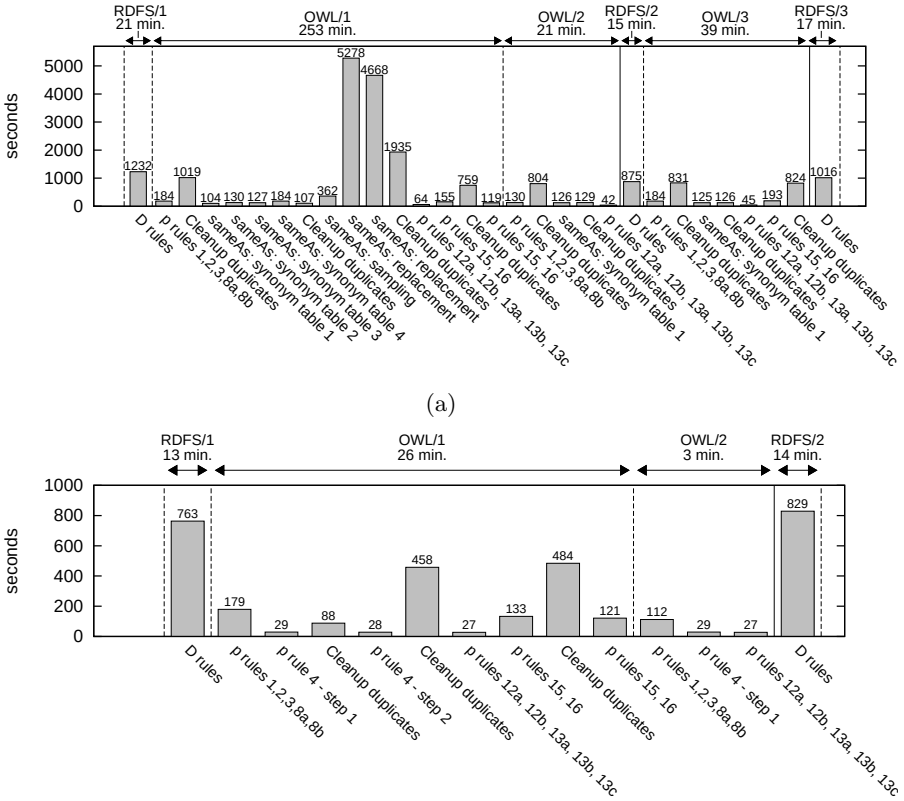
The evaluation was performed using three datasets: *UniProt* is one of the largest (1.51 billion unique triples) curated sets of real-world OWL statements available to date, and has been used before to stress-test the performance of OWL reasoners. *LDSR* includes DBPedia, Freebase, Geonames and other datasets representing general knowledge and consists of 0.9 billion triples. *LUBM* is a widely used benchmark that can generate semi-realistic datasets of arbitrary size. We have chosen to use LUBM because: (a) it is widely used for reasoner evaluation, allowing comparison of our results with existing approaches; (b) there exists no real-world dataset of the size we want to test (up to 100 billion triples); (c) reasoning over arbitrary triples retrieved from the Web would result in useless and unrealistic derivations [7]. All datasets do indeed use the OWL Horst fragment.

4.2 Experimental results

Performance on real world data: For UniProt, our system derived 2.03 billion triples, as well as a synonyms table for `owl:sameAs` relationships, consisting of equality statements between 35 million entries. The entire process took 6.1 hours on 32 nodes. For LDSR, our system derived 0.94 billion triples in 3.52 hours. Figure 2a shows the sequence of the launched MapReduce jobs for UniProt along with the time spent for each of them. The analysis shows that the computation is dominated by the costs of the equality reasoning, in particular, the costs of replacing all RDF resources by the canonical members of the `sameAs`-equivalence classes. When no `owl:sameAs` statements are present (as in the LUBM case, figure 2b), the costs are more evenly spread across the different phases. A future challenge is to reduce the cost of the equality-reasoning, either by smarter algorithms or by further parallelization of this phase.

Scalability: Table 3a shows how our approach scales with an increasing number of compute nodes, using 10 billion triples generated by LUBM as a fixed input. We define speedup as $\frac{runtime\ for\ baseline}{runtime}$. We use the time on the 8-node configuration as baseline, because in DAS3 a single node does not have enough

[5] `http://www.ontotext.com/owlim/`
[6] `http://www.cs.vu.nl/das3/`



(a)

(b)

Fig. 2. Execution steps for UniProt (a) and LUBM (b)

resources (disk space) to store all the data. Table 3b shows how our approach scales with increasing input size, using a fixed configuration of 64 nodes.

We make the following observations:

- The throughput is significantly (almost 30%) higher for larger datasets. This is attributed to platform startup overhead which is amortized over a larger processing time for large datasets. The platform overhead is also responsible for the superlinear speedup in table 3a. Since the overhead becomes more relevant with fewer nodes, the calculated speedup will be higher than the real one.
- The execution time greatly depends on the complexity of the input: on UniProt and LDSR, we achieve a throughput of 68.3 Ktriples/sec and 74.9 Ktriples/sec respectively, while the throughput on the much simpler LUBM dataset is around 10 times higher (table 3b). If the input is more complex, the algorithm needs to launch more iterations to reach fixpoint (as shown in Figure 2).

Nodes	Runtime (hours)	Speedup
8	44.4	1.00
16	22.3	1.99
32	10.6	4.17
64	5.0	8.78

(a)

Input (GTriples)	Output (MTriples)	Runtime (hours)	Throughput (Kt/sec)
1.07	495.5	0.61	455.2
10.71	4971.7	4.06	684.6
102.50	47563.1	45.77	606.8

(b)

Fig. 3. Scalability over the number of nodes (left) and over input data (right)

– Considering the effects of the platform overhead, we conclude that the results show linear scalability regarding the size of the input and number of nodes. Currently, the only way to test this is by using the LUBM benchmark data. We have no evidence on how the algorithm behaves with a real world dataset of 100 billion triples, since such dataset currently does not exist.

5 Related work

Hogan *et al.* [6] compute the closure of an RDF graph using two passes over the data on a single machine. They implement only a fragment of the OWL Horst semantics to allow efficient materialization and to prevent "ontology hijacking".

Schlicht and Stuckenschmidt [13] show peer-to-peer reasoning for the expressive ALC logic but focusing on distribution rather than performance.

Soma and Prasanna [14] present a technique for parallel OWL inferencing through data partitioning. Experimental results show good speedup but only on very small datasets (1M triples) and runtime is not reported. In contrast, our approach needs no explicit partitioning phase and we show that it is scalable over increasing dataset size.

In [10, 12], we have presented a technique based on data-partitioning in a peer-to-peer network. A load-balanced auto-partitioning approach was used without upfront partitioning cost. Experimental results were however only reported for datasets of up to 200M triples.

In Weaver and Hendler [16], straightforward parallel RDFS reasoning on a cluster is presented. This approach replicates all schema triples to all processing nodes and distributes instance triples randomly. Each node calculates the closure of its partition using a conventional reasoner and the results are merged. To ensure that there are no dependencies between partitions, triples extending the RDFS schema are ignored. This approach is not extensible to richer logics, or complete RDFS reasoning, since, in these cases, splitting the input to independent partitions is impossible.

Newman *et al.* [11] decompose and merge RDF molecules using MapReduce and Hadoop. They perform SPARQL queries on the data but performance is reported over a dataset of limited size (70,000 triples).

Several proposed techniques are based on deterministic rendezvous-peers on top of distributed hashtables [1, 2, 4, 9]. However, because of load-balancing problems due to the data distributions, these approaches do not scale [10].

Some Semantic Web stores support reasoning and scale to tens of billions of triples[7]. We have shown inference on a triple set which is one order of magnitude *larger* then reported anywhere (100 billion triples against 12 billion triples). Furthermore, our inference is 60 times *faster* (10 billion triples in 4 hours against 12 billion triples in 290 hours for LUBM) against the best performing reasoner (BigOWLIM). For UniProt, BigOWLIM 3.1 needs 21 hours to perform forward reasoning on 1.15 billion triples[8] (yielding a throughput of 15.2 Ktriples/sec) while our system needs only 6 hours for 1.5 billion triples (yielding a throughput of 68.3 Ktriples/sec). It should be noted that the comparison of our system with RDF stores is not always meaningful, as our system does not support querying.

6 Conclusion

Summary In this paper, we have shown a massively scalable technique for parallel OWL Horst forward inference and demonstrated inference over 100 billion triples. Both in terms of processing throughput and maximum data size, our technique outperforms published approaches by a large margin.

Discussion of scope The computational worst-case complexity of even the OWL Horst fragment precludes a solution that is efficient on all inputs. Any approach to efficient reasoning must make assumptions about the properties of realistic datasets, and optimize for those realistic cases. Some of the key assumptions behind our algorithms are: (a) The schema must be small enough to fit in main memory; (b) for rules with multiple joins, some of the joins must be performed in-memory, which could cause memory problems for some unrealistic datasets or for machines with very limited memory; (c) we assume that there is no ontology hijacking [7]; and (d) all the input is available locally in the distributed filesystem. The difference in performance on UniProt and LUBM shows that the complexity of the input data strongly affects performance Although it is easy to create artificial data which break the performance, we did not observe such cases in realistic data. In fact, the above assumptions (a)-(d) could also serve as guidelines in the design of ontologies and datasets, to ensure that they can be used effectively.

Future challenges The technique presented is optimized for the OWL-Horst rules. Future work lies in reasoning over user-supplied rulesets, where the system would chose the correct implementation for each rule and the most efficient execution order, depending on the input.

Furthermore, as with all scalable triple stores, our approach cannot efficiently deal with distributed data. Future work should extend our technique to deal with data streamed from remote locations.

[7] `http://esw.w3.org/topic/LargeTripleStores`
[8] D5.5.2 at `http://www.larkc.eu/deliverables`

In fact, we believe that this paper establishes that computing the closure of a very large centrally available dataset is no longer an important bottleneck, and that research efforts should switch to other modes of reasoning. Query-driven backward-chaining inference over distributed datasets might turn out to be more promising than exhaustive forward inference over centralized stores.

References

[1] D. Battré, A. Höing, F. Heine, and O. Kao. On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores. In *Proceedings of the VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*. 2006.

[2] M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proc. of the WWW 2004*.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of USENIX OSDI*, pp. 137–147. 2004.

[4] Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable distributed ontology reasoning using DHT-based partitioning. In *Proceedings of the Asian Semantic Web Conference (ASWC)*. 2008.

[5] P. Hayes, (ed.) *RDF Semantics*. W3C Recommendation, 2004.

[6] A. Hogan, A. Harth, and A. Polleres. Scalable authoritative OWL reasoning for the web. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

[7] A. Hogan, A. Polleres, and A. Harth. Saor: Authoritative reasoning for the web. In *Proceedings of the Asian Semantic Web Conference (ASWC)*. 2008.

[8] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2–3):79–115, 2005.

[9] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on top of DHTs. In *Proceedings of the ISWC 2008*.

[10] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the data skew: Distributed inferencing by speeddating in elastic regions. In *Proc. of the WWW 2010*.

[11] A. Newman, Y. Li, and J. Hunter. Scalable semantics the silver lining of cloud computing. In *Proceedings of the 4th IEEE International Conference on eScience*. 2008.

[12] E. Oren, S. Kotoulas, G. Anadiotis, *et al.* MARVIN: distributed reasoning over large-scale Semantic Web data. *Journal of Web Semantics*, 2009.

[13] A. Schlicht and H. Stuckenschmidt. Peer-to-peer reasoning for interlinked ontologies. In *to appear in the International Journal of Semantic Computing, Special Issue on Web Scale Reasoning*. 2010.

[14] R. Soma and V. Prasanna. Parallel inferencing for OWL knowledge bases. In *International Conference on Parallel Processing*, pp. 75–82. 2008.

[15] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the ISWC '09*. 2009.

[16] J. Weaver and J. Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In *Proceedings of the ISWC '09*. 2009.