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

Contents lists available at ScienceDirect

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

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

WebPIE: A Web-scale Parallel Inference Engine using MapReduce

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

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

ARTICLE INFO

Article history:
Available online 22 July 2011

Keywords:
Semantic Web
MapReduce
High performance
Distributed computing
Reasoning

ABSTRACT

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

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

1. Introduction

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

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

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

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

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

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

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

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

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

1570-8268/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.websem.2011.05.004

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

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

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

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

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

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

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

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

2. The MapReduce programming model

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

2.1. A simple MapReduce example: term count

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

Algorithm 1. Counting term occurrences in RDF NTriples files

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

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

2.2. Characteristics of MapReduce

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

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

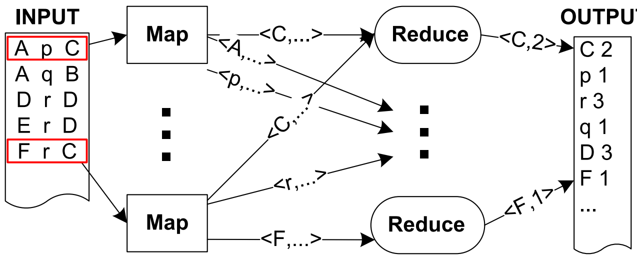


Fig. 1. MapReduce processing.

- the *reduce* operates on all pieces of data that share the same key. By assigning proper keys to data items during the *map*, 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 term *x* will be slower than others. To use MapReduce efficiently, we must find balanced partitions of the data;
- because the data resides on local nodes, and the physical location of data is known, the platform performs *locality-aware scheduling*: if possible, *map* and *reduce* are scheduled on the machine holding the relevant data, moving computation instead of data.

3. RDFS reasoning with MapReduce

The closure under the RDFS semantics [14] can be computed by applying all RDFS rules iteratively on the input until no new data is derived. Rules with one antecedent can be applied with a single pass on the data. Rules with two antecedents are more challenging to implement since they require a join over two parts of the data: if the join is successful, we derive a consequence.

The rest of this section explains in detail how we can efficiently apply the RDFS rules with MapReduce. First, in Section 3.1, we describe how to implement an example rule in MapReduce in a straightforward way. Next, we describe the problems that arise with such a straightforward implementation and we discuss some optimizations that we introduce to solve, or at least reduce, such problems.

3.1. Example rule execution with MapReduce

Applying one rule means performing a join over some terms in the input triples. Let us consider rule 9 from Table 1, which derives `rdf:type` based on the sub-class hierarchy. This rule contains a single join on variable *x*, and it can be implemented with a *map* and *reduce* function, as shown in Fig. 2 and Algorithm 2.

Algorithm 2. Naive sub-class reasoning (RDFS rule 9)

```
map(key, value):
  // key: linenumber (irrelevant)
  // value: triple
  switch triple.predicate
  case "rdf:type":
    emit(triple.object, triple); // group (s rdf:type x) on x
  case "rdfs:subClassOf":
    emit(triple.subject, triple); // group (x rdfs:subClassOf y)
    on x

reduce(key, iterator values):
  // key: triple term, eg x
  // values: triples, eg (s type x), (x subClassOf y)
  superclasses=empty;
  types=empty;

  // we iterate over triples
  // if we find subClass statement, we remember the super-
  classes
  // if we find a type statement, we remember the type
  for (triple in values):
    switch triple.predicate
    case "rdfs:subClassOf":
      superclasses.add(triple.object) // store y
    case "rdf:type":
      types.add(triple.subject) // store s

  for (s in types):
    for (y in classes):
      emit(null, triple(s, "rdf:type", y));
```

Table 1
RDFS rules (*D*)[14].

1	*s p o* (if *o* is a literal)	⇒ _:n rdf:type rdfs:Literal
2	*p* rdfs:domain *x* & *s p o*	⇒*s* rdf:type *x*
3	*p* rdfs:range *x* & *s p o*	⇒ *o* rdf:type *x*
4a	*s p o*	⇒ *s* rdf:type rdfs:Resource
4b	*s p o*	⇒ *o* rdf:type rdfs:Resource
5	*p* rdfs:subPropertyOf *q* & *q* rdfs:subPropertyOf *r*	⇒ *p* rdfs:subPropertyOf *r*
6	*p* rdf:type rdf:Property	⇒ *p* rdfs:subPropertyOf *p*
7	*s p o* & *p* rdfs:subPropertyOf *q*	⇒ *s p o*
8	*s* rdf:type rdfs:Class	⇒ *s* rdfs:subClassOf rdfs:Resource
9	*s* rdf:type *x* & *x* rdfs:subClassOf *y*	⇒ *s* rdf:type *y*
10	*s* rdf:type rdfs:Class	⇒ *s* rdfs:subClassOf *s*
11	*x* rdfs:subClassOf *y* & *y* rdfs:subClassof *z*	⇒ *x* rdfs:subClassOf *z*
12	*p* rdf:type rdfs:ContainerMembershipProperty	⇒ *p* rdfs:subPropertyOf rdfs:member
13	*o* rdf:type rdfs:Datatype	⇒ *o* rdfs:subClassOf rdfs:Literal

In the *map* function, we process each triple and output a key/value pair, using as value the original triple, and as key the triple's term (`s,p,o`) on which the join should be performed. To perform the join, triples with `rdf:type` should be grouped on their object (eg. "x"), while triples with `rdfs:subClassOf` should be grouped on their subject (also "x"). 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 join.

One execution of these two functions will not find *all* corresponding conclusions because there could be some derived statements which trigger other joins. For example, in order to compute the transitive closure of a chain of *n* `rdfs:subClassOf`-inclusions, we would need to apply the above *map/reduce* steps $\log_2 n$ times.

The algorithm we presented encodes only rule 9 of the RDFS rules. We would need to add other, similar, algorithms to implement each of the other rules. These other rules are interrelated: one rule can derive triples that can serve as input for another rule. For example, rule 2 derives `rdf:type` information from `rdfs:domain` statements. After applying that rule, we would need to re-apply our earlier rule 9 to derive possible superclasses. Thus, to produce the complete RDFS closure of the input data using this technique we need to add more *map/reduce* functions and keep executing them until we reach a fixpoint.

3.2. Problems of RDFS reasoning with MapReduce

The previously presented implementation is straightforward, but inefficient. The problems are:

Derivation of duplicates. We encoded, as example, only rule 9 and we launched a simulation over the Falcon dataset, which contains 35 million triples. After 40 min the program had not yet terminated, but had already generated more than 50 billion triples and filled our disk space. Considering that the unique derived triples from Falcon are no more than 1 billion, the ratio of unique derived triples to duplicates is at least 1:50. With such duplicates ratio, this implementation cannot scale to a large size because the communication and storage layers will fail to store the additional data. This is a challenge concerning *large data transfers*, as mentioned in the introduction.

Join with schema triples. If we execute the joins as described in the example, there will be some groups which will be consistently larger than others and the system will be unable to parallelize efficiently the computation, since a group has to be

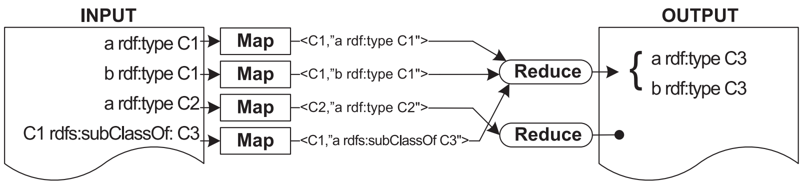


Fig. 2. Encoding RDFS rule 9 in MapReduce.

processed by one single node. This problem is an instance of the challenges concerning *load balancing*.

Fixpoint iteration. In order to compute the closure, we need to keep applying the rules until we finish deriving new information. The number of iterations depends on the complexity of the input (see the challenge class *reasoning complexity*). Nevertheless, we can research whether a specific execution order leads to less iterations than another.

In the next subsections, we introduce three optimizations that address these three problems to greatly decrease the time required for the computation of the closure. These optimizations are: (i) limit the number of duplicates with a pre-processing step; (ii) optimize the joins by loading the schema triples in memory; and (iii) minimize the number of iterations needed by ordering the execution of rules according to their dependencies.

3.3. Optimization: loading schema triples in memory

Typically, schema triples are by far less numerous than instance triples [15]. Consider the Billion Triple Challenge dataset, which is a dataset of data crawled from the Web. This dataset was built to be a realistic representation of the Semantic Web and therefore can be used to infer statistics which are valid for the entire web of data. The results, shown in Table 2, confirm this assumption and allow us to exploit this fact to execute the joins more efficiently.

Before we continue our explanation, we must note that all the RDFS rules that require a join have at least one schema triple as antecedent. This means that all the RDFS joins are either between two schema triples or between one schema triple and one instance triple. We can exploit this fact to load all schema triples in memory and perform the join with the input triples in a streaming fashion.

As an illustration, let us consider rule 9 of Table 1 The set of `rdf:type` triples is typically far larger than the set of `rdfs:subClassOf` triples. Essentially, what we do is to load the small set of `rdfs:subClassOf` triples in memory and launch a MapReduce job that joins the many instance triples it receives as input (the `rdf:type` triples) with the in-memory schema triples. This methodology is different from the straightforward implementation where we were grouping the antecedents during the map phase and performing the join during the reduce phase (see the example in Section 3.1).

The advantages of performing a join against in-memory schema triples are: (i) we do not need to repartition our data in order to perform the join, meaning that we are reducing *data transfers*; (ii) for the same reason, the *load balance* is perfect, since any node may process any triple from the input; and (iii) we can calculate the closure of the schema in-memory, avoiding the need to iterate $\log_2 n$ times over our input, in practically all cases.

The main disadvantage of this method is that it works only if the schema is small enough to fit in memory. We have shown that this holds for generic web data but there might be some specific contexts in which this assumption does not hold anymore.

Table 2
Schema triples (number and fraction of total triples) in the Billion Triple Challenge dataset.

Schema type	Number	Fraction (%)
Domain, range (p rdfs:domain D, p rdfs:range R)	30.000	0.004
Sub-property (a rdfs:subPropertyOf b)	70.000	0.009
Sub-class (a rdfs:subClassOf b)	2.000.000	0.2

3.4. Optimization: data preprocessing to avoid duplicates

The join with the schema triples can be executed either during the map phase or during the reduce phase. If we execute it during the map phase, we can use the reduce phase to filter out the duplicates.

Let us take, as an example, rule 2 (`rdfs:domain`) of the RDFS ruleset. Assume we have an input with ten different triples that share the same subject and predicate but have a different object. If the predicate has a domain associated with it and we execute the join in the mappers, the framework will output a copy of the new triple for each of the ten triples in the input. These triples can be correctly filtered out by the reducer, but they will cause significant overhead because they will need to be stored locally and be transferred over the network.

After initial experiments, we have concluded that the number of duplicates generated during the map phase was too high and it was affecting the overall performance. Therefore, we decided to move the execution of the join to the reduce phase and use the map phase to preprocess the triples.

To see how it works, let us go back to rule 2. Here, we can avoid the generation of duplicates if we first group the triples by subject and then execute the join over the single group. Grouping triples by value means that all derivations involving a given *s* will be made at the same reducer, making duplicate elimination trivial.

This methodology can be generalized for other joins by setting the parts of the input triples that are also used in the derived triple as the intermediate key and the part that should be matched against the schema as value. These parts depend on the applied rule. In the example above, the only part of the input that is also used in the output is the subject while the predicate is used as matching point. Therefore, we will set the subject as key and the predicate as value. Since the subject appears also in the derived triple, it is impossible that two different groups generate the same derived triple. Eventual duplicates that are generated within the group can be filtered out by the reduce function with the result that no duplicates can be generated.

This process does not avoid all duplicate derivations because we can still derive duplicates against the original input. Looking back at rule 2, when the reduce function derives that the triple's subject is an instance of the predicate's domain, it does not know whether this information was already explicit in the input. Therefore, we still need to filter out the derivation, but only against the input and not between the derivations from different nodes.

3.5. Optimization: ordering the application of the RDFS rules

We have analyzed the RDFS ruleset to understand which rule may be triggered by which other rule. By ordering the execution of rules according to their data dependencies, we can limit the number of iterations needed to reach full closure.

Rules 1, 4, 6, 8, 10 are ignored without loss of generality. These rules have one antecedent and can be implemented at any point in time with a single pass over the data and the outcome they produce cannot be used for further non-trivial derivation.

Also rules 12 and 13 have a single antecedent and therefore can be implemented with a single pass over the input. However, their outcome can be used for further non-trivial derivation and therefore we must include them in our discussion.

First, we have categorized the rules based on their output:

- rules 5 and 12 produce schema triples with *rdfs:subPropertyOf* as predicate;
- rules 11 and 13 produce schema triples with *rdfs:subClassOf* as predicate;
- rules 2, 3,and 9 produce instance triples with *rdf:type* as predicate;
- rule 7 may produce arbitrary triples.

We also have categorized the rules based on the predicates in their antecedents:

- rules 5 and 11 operate only on triples with *rdfs:subClassOf* or *rdfs:subPropertyOf* in the predicate position;
- rules 9, 12 and 13 operate on triples with *rdf:type*, *rdfs:subClassOf* or *rdfs:subPropertyOf* in the predicate position;
- rules 2, 3 and 7 can operate on arbitrary triples.

Fig. 3 displays the relation between the RDFS rules based on their input and output (antecedents and consequents). Arrows signify data dependencies. An ideal execution should proceed bottom up: first apply the sub-property rules (rules 5 and 7), then rules 2 and 3, and finally rules 9, 11, 12 and 13.

Rules 12 and 13 can produce triples that serve the rules at the lower levels. However, looking carefully, we see that these connections are possible only in few cases: The output of rule 12 can be used to fire rules 5 and 7. In order to activate rule 5, there must be either a superproperty of `rdfs:member` or a subproperty of *p*. In order to activate rule 7 there must be some resources connected by *p*. We ignore the first case of rule 5, following the advice against “ontology hijacking” from [15] that suggests that users may not redefine the semantics of standard RDFS constructs. The other two cases are legitimate but we have noticed that, in our experiments, they never appear.

The same discussion applies to rule 13 which can potentially fire rules 9 and 11. Again, we have noticed that the legitimate cases when this happens are very rare.

The only possible loop we can encounter is when we extend the schema by introducing subproperties of `rdfs:subproperty`. In this case rule 7 could fire rule 5, generating a loop. Although not disallowed, we have never encountered such case in our experiments.

Given these considerations, we can conclude that there is an efficient rule ordering which consistently reduces the number of times we need to apply the rules. The cases where we might have to reapply the rules either do not require expensive computation (for rules 12 and 13 an additional single pass over the data is enough) or occur very rarely or not at all.

Therefore, as our third optimization, we apply the rules as suggested in Fig. 3, and launch an additional single-pass job to derive the eventual derivation triggered by rules 12 and 13. The loop generated by schema extension does not occur in our data, and does not seem to be a real issue on real world datasets. However, in case it happens, all the rules must be re-executed until fix closure.

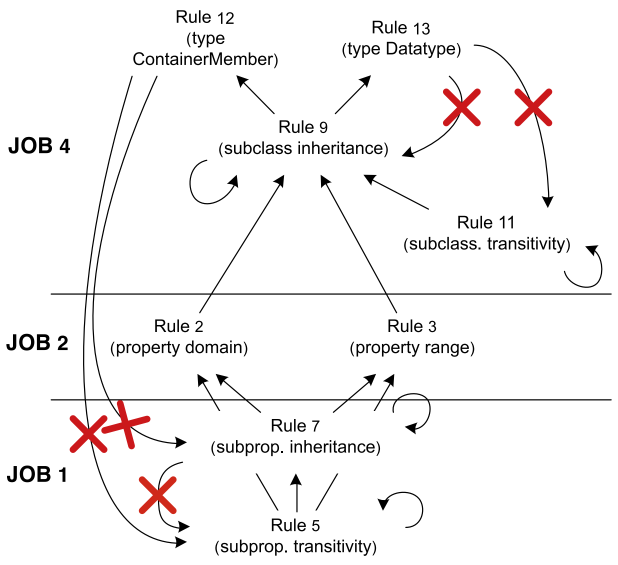


Fig. 3. Relation between the various RDFS rules. The red cross indicates the relations that we do not consider.

4. OWL reasoning with MapReduce

In the previous section, we have described how to perform efficient reasoning under the RDFS semantics. Here, we move to the more complex OWL semantics considering the ruleset of the ter Horst fragment [18]. The reasons of choosing this fragment are: (i) it is a *de facto* standard for scalable OWL reasoning, implemented by industrial strength triple stores such as OWLIM; (ii) it can be expressed by a set of rules; and (iii) it strikes a balance between the computationally infeasible OWL full and the limited expressiveness of RDFS. The OWL Horst ruleset consists of the RDFS rules [14], shown in Table 1 and the rules shown in Table 3.

As with the RDFS ruleset, we omit some rules with one antecedent (rules 5a and 5b). These rules can be parallelized efficiently and are commonly ignored by reasoners since they yield consequences that can be easily calculated at query-time. The other rules introduce new challenges.

In the remaining of this section we will describe these additional challenges and we will propose a new set of optimizations to address them. Again, we will defer thorough explanation of the algorithms to Appendix A.2.

4.1. Challenges with OWL reasoning with MapReduce

Some of the rules in Table 3 can be efficiently implemented using the optimizations presented for the RDFS fragment. These are rules *3, 8a, 8b, 12a, 12b, 12c, 13a, 13b, 13c, 14a, and 14b*.

The rest of the rules introduce new challenges:

Joins between multiple instance triples. In RDFS, at most one antecedent can be matched by instance triples. In this fragment, 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.

Exponential number of derivations. This problem is evident with the *sameAs* derivations and we illustrate it with a small example. Consider that we have one resource which is used in

Table 3
OWL Horst rules.

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

n triples. When we add one synonym of this resource (which is also used in *n* triples), rule 11 would derive $2^1 \cdot n$ triples. If there are 3 synonyms, the reasoner will derive $2^3 \cdot n$ new resources and for *L* synonyms, it would derive $2^l \cdot n$. The I/O system can not sustain this exponential data growth, and soon enough, it would become a performance bottleneck.

Multiple joins per rule. In RDFS, all the rules require at most one join between two antecedents. Here, rules 11, 15 and 16 require two joins. For example, rule 11 requires a join between `u p v` and u owl:sameAs x on `u` and a join between `u p v` and `v owl:sameAs y` on `v`;

Fixpoint iteration. This problem was also present in the RDFS ruleset, but there, by making some assumptions, we could identify an execution order with no loops. This is not the case of OWL and we are required to iterate until fixpoint.

We propose four new optimizations that focus on each of these problems: (i) we limit the number of duplicates exploiting the characteristics of the rules. (ii) We build a synonyms table trefo avoid the materialization of the `sameAs` derivations. (iii) We execute part of the joins in memory when we have to perform multiple joins. (iv) We propose execution strategies to limit the number of times we execute the rules.

In the remaining of this section we describe each optimization in more detail.

These optimizations cannot be generically applied to any rule as it was the case with the RDFS fragment, but instead are tailored to the characteristics of individual rules or groups of rules in the ter Horst fragment. Even though some of the optimizations presented here might be generalized, additional rules that are not part of the ter Horst fragment (for example some that are part of OWL 2 RL) will require new optimizations to solve the challenges they pose.

4.2. Optimization: limit duplicates when performing joins between instance triples

In MapReduce, a join between instance triples can be performed only in the reduce phase, as shown in the example in Section 3.1. This execution can potentially lead to load balancing problems and exponential derivation of duplicates. In this subsection, we propose a series of techniques to avoid (or limit) the number of duplicate derivations.

The rules which require joins between instance triples are rules `1`, `2`, `4`, `7`, `11`, `15` and `16`. We will defer discussion on the last four rules to Sections 4.3 and 4.4.

Rules 1 and 2 require a join between any two triples; however, if we look more carefully, we notice that the joins are on two resources, not on one, and the generic triples are filtered using information in the schema. In rule 1, the join is on the subject and predicate, while in rule 2 it is on the predicate and the object. Because we group on two resources, it is very unlikely that the reduce groups will be big enough to cause load balancing problems.

Rule 4 requires a three-way join between one schema triple and two instance triples. For readability we repeat the rule below:

```
if p rdf:type owl:TransitiveProperty
and u p w
and w p v
then u p v
```

At first sight, rule 4 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. However, 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 in a straightforward way will lead to a large number of duplicates, because every time the rule is applied, the same relationships will be inferred. For a transitive property chain of length *n*, a straightforward 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. For example, consider the following three triples: `a p b`, `b p c` and `c p d` where `a`,`b`, and `c` are generic resources and `p` is a transitive property. Fig. 4 graphically represents the corresponding RDF graph. Applying rule 4 consists of calculating the transitive closure of such chain; i.e. explicitly connect `a` with `c` and `d` and connect `b` with `d`.

With one MapReduce job we are able to perform a join only between triples with a shared resource. Therefore, we need $\log_n$ MapReduce jobs to compute the transitive closure of a chain of length *n*.

We define the *distance* between two subsequent resources in a transitive chain as the number of hops necessary to reach the second from the first one. In our example the terms `a` and `b` have distance one while the terms `a` and `c` have distance two because they are connected through `b`. When we execute the *n*th MapReduce job we would like not to derive triples which have a graph distance less than or equal to 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}.

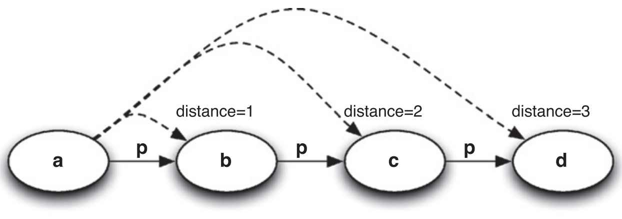


Fig. 4. OWL transitive closure.

In the ideal case, we completely avoid duplicates. Nevertheless, when there are different chains that intersect, the algorithm will still produce duplicate derivations, but much fewer than without this optimization. This algorithm is described in pseudocode in Appendix A.2 and Algorithm 9.

4.3. Optimization: build sameAs table to avoid exponential derivation

To avoid exponential derivation, 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 3) and we always refer to the id of the sets. In other words, we choose a nominal representation for equivalent resources under `owl:sameAs`. This is common practice in industrial strength triple stores, and dealing with such a structure is much more efficient at query time. 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 number of intermediate data that need to be processed and the size of the output.

This process makes rules 6, 9, 10 redundant, since the results of their application will be merged using the synonyms table.

Nevertheless, we still need to apply the logic of rules 7 and 11. We report them below:

Rule 7	Rule 11
if v owl:sameAs w and w owl:sameAs u then v owl:sameAs u	if u p v and u owl:sameAs x and v owl:sameAs y then x p y

Both rules are problematic because they involve a two-way and a three-way join between instance triples. As before, and because the join is on instance data, we cannot load one side in memory. Instead we are obliged to perform the join by partitioning the input on the part of the antecedents involved in the join.

Again, such an 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 because 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 that are present in the datasets (i.e. resources connected by the `owl:sameAs` relation) 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 apply the same logic of rule 11, building a new dataset where we only refer to the single canonical representation for each equivalence class.

We will now describe how we can build the `sameAs` table and replace all items by their canonical representation using MapReduce.

4.3.1. Building the sameAs table

Our purpose is to group the resources connected with the `sameAs` relationship and assign a unique ID to each group.

For example, suppose that we have four triples: `a sameAs b`, `b sameAs c`, `c sameAs d` and `x sameAs y`. These triples can be grouped in two groups: one containing `a`, `b`, `c` and the other containing `x` and `y`. We would like to assign a unique ID to each group and store this information in a table so that later on we can replace every instance of the resources (a, b, etc.) with the corresponding group ID.

Since the `owl:sameAs` relation is symmetric, the corresponding RDF graph of the `sameAs` triples is undirected (`a sameAs b` implies `b sameAs a`). What we do is to assign a unique ID to each resource and turn the undirected graph into a directed one by making the edges point to the node with higher ID. The node with the lowest ID will be the group's ID that will represent all the other nodes that are reachable from it.

Some nodes can be reached only through several hops. For example, suppose that `a` is the resource in our group with the lowest ID. The resource `d` can be reached only through `b` and `c`. In order to efficiently calculate all nodes reachable from the one with the lowest ID, we have developed a MapReduce algorithm (reported in pseudocode in Appendix A, Algorithm 10) where we incrementally create shorter paths starting from the node with the lowest ID removing all the intermediate edges that are no longer needed. The algorithm will terminate when all edges will originate from the node with the lowest ID and there will not be any intermediate one. Looking back at our example, the output of this algorithm would consist of the triples: `a sameAs b`, `a sameAs c`, `a sameAs d` and `x sameAs y`. These triples will form the sameAs table since they contain the relation between each sameAs resource (e.g. `a`, `b`, `c`, etc.) and their corresponding group key (e.g. `a`).

4.3.2. Replacing resources with their canonical representation

Since our purpose is to replace in the original dataset the resources in the `sameAs` table with 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 but such approach suffers from a severe load balancing problem since the term distribution is very uneven. 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 (similar to our technique for performing joins between schema and instance triples in RDFS). In our implementation, we typically sample a random subset of 1% 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 because 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.

4.4. Optimization: perform redundant joins to avoid load balancing problems

Rules 15 and 16 are challenging because they require multiple joins. Both rules are reported below:

Rule 15	Rule 16
if v owl:someValuesFrom w	if v owl:allValuesFrom u
and v owl:onProperty p	and v owl:onProperty p
and u p x	and w rdf:type v
and x rdf:type w	and w p x
then u rdf:type v	then x rdf:type u

In this section, we only discuss rule 15. The discussion for rule 16 is entirely analogous.

Rule 15 requires three joins on the antecedents' patterns. Two of them can be classified as schema patterns. These are the triples of the form v `owl:someValuesFrom` w and v `owl:onProperty` p. Since the schema is typically small, we can load both patterns in each node's main memory and perform the join between them using standard techniques.

Still, there are two joins left to perform. The first join will be between a small set (the result of the in-memory join) and a large one (the "type" triples). This join will produce a large set of triples that needs to be further joined with the rest of the input (the $u\ p\ x$ pattern).

One option is to perform one of the remaining joins during the map phase and the other during the reduce. In such a scenario, the product of the first join is loaded in the nodes' main memory and the second join is performed against the "type" triples that are read in the input. When this join succeeds, the triples are forwarded to the reduce phase so that the third join against the remaining triples can be performed in the classical way.

There are two problems with this approach. First, it requires that one of the two instance patterns (either the "type" triples or the generic one) is always forwarded to the reduce phase to be matched during the third join. Second, the third join would be performed on a single resource and this will generate load balancing problems.

To solve these two problems, we perform the join between the schema and both instance patterns during the map phase. This means that the "type" triples will be matched on their objects (w) while all the other triples will be matched on the predicate (p). After that, we partition the triples not only on the common resource (x), but also on the common resource between the schema (v) that we can easily retrieve from the in-memory schema.

The advantages of this technique are: (i) we filter both instance patterns so that fewer triples will be forwarded to the reduce phase and (ii) we partition the triples on two resources instead of one, practically eliminating the load balancing issue.

The disadvantages are that we perform more joins than needed; however, the joins are performed against an in-memory data structure and therefore they do not introduce significant overhead in the computation.

4.5. Optimization: execution strategies

Executing one rule on a very large input is a time consuming operation that should be done only if strictly necessary. In RDFS we found a specific execution order which minimizes the number of rule executions. Unfortunately, such order does not exist for OWL and we have to execute the rules until all of them stop deriving new statements.

In this section, we propose a set of execution strategies that dynamically adapt to the complexity of the input, reducing the number of job executions.

Two types of strategies are used:

Rule execution strategies, defining the order in which the rules are executed based on the derivation that they produce.

Duplicate elimination strategies, deciding when to launch jobs to clean up derived duplicates.

Table 4
Rule execution strategies (RES).

Fixed	Rules are executed in a fixed order until all of them do not derive anything
Greedy	With this strategy first we execute the rules that previously had derived *most* of the statements leaving the others for later. The intuition behind this strategy is that we want to maximize the number of derivations for every execution and the assumption is that past derivation is indicative of future derivation

Table 5
Duplicate elimination strategies (DES).

Always	The duplicates are eliminated immediately after every rule has derived something. This strategy aims at minimizing the data that needs to be processed by the future jobs by eliminating duplicates as soon as possible.
End	Duplicate derivation is deferred to the end, aiming at reducing the cost for performing duplicate elimination
Threshold	The cleaning job is launched only if the derivation is large enough to justify the additional computation. The limit is set by a threshold

The first type of strategies addresses the problem of executing the right rule so that we can reduce the number of executions. For example, if the dataset contains no `sameAs` statements, we can safely avoid executing the `sameAs` rules unless some other rules produce some `sameAs` derivation.

The second type of strategies addresses the problem of duplicate elimination. For some rules, after execution, we are required to launch an additional job to filter eventual duplicate derivation. This additional job is conceptually very simple: it first reads all the triples, the ones in the input and the ones derived so far, and then it outputs only the new derived ones.

This additional job is computationally expensive because it needs to process all the input even in the case where there is a small derivation. Consider the case when the execution of a rule produced one new statement. Because this new statement might be a duplicate, we need to process the entire input to ensure that it is unique.

We propose two rule execution strategies (RES) and three duplicate elimination strategies (DES), which are described in Tables 4 and 5 respectively. These strategies are based on heuristics and their performance depends on the input.

5. Incremental reasoning

In some use-cases, there is an initial dataset that is updated and performing the reasoning only on the part that is added is more efficient. The MapReduce programming model was not originally designed to efficiently perform small and frequent updates, but to process a large input offline, where response time is not an issue. The reasoning algorithms in [30] were unable to handle incremental reasoning, forcing full recomputation at every update.

To support efficient incremental reasoning, we need to minimize recomputation. This can be achieved with the following: first, we distinguish between statements that were already existing and the added ones. Secondly, we adapt the algorithms to process the statements according to their status (old or new) and infer the new information only if it is related to the new ones.

For the former, we store next to each triple the time when the triple was added to the dataset. In this way we are able to recognize whether a triple was already present or it was just inserted.

The latter is more challenging and requires changing the MapReduce algorithms. The general policy to is to allow the join only if at least one of the antecedents has a timestamp higher than the last time that same rule was executed. In order to limit the I/O as much as possible, we perform this filtering as soon as possible. For example, in case of an RDFS rule, we can check the timestamps either during the map phase when we group the triples, or during the reduce phase when we perform the join. If we perform this check during the map phase, we avoid that the triples are sent to the reducers, and the associated data transfer cost.

It is obvious that the performance of incremental updates is highly dependent on the input data. As we will show in the evaluation, the performance is not proportional to the size of the update but more on the degree it influences the existing data. For example, if we add `sameAs` triples to a very large dataset, then the `sameAs` table will need to be updated. These updates will have to be reflected in the input, which is a very costly process. On the other side, if the added triples that do not contain information that change the rest of the input, the reasoner will not need to perform any reasoning with the original data and the execution will be relatively fast.

6. Evaluation

In the previous sections we have described the problems of large scale reasoning and a number of optimizations as solutions to them. We have implemented a prototype and in this section we will evaluate its performance.

In our case performing an isolated evaluation of our optimizations is difficult because they cannot be evaluated independently to assess their real gain. In fact, without these optimizations the system will crash due to the problems they address (excessive duplicates derivation, etc.) so that an individual evaluation of the optimizations is technically impossible. Since we are obliged to evaluate the system as a whole, we analyzed the performance varying the parameters that might influence the reasoning performance. For example, we tried to launch the reasoner on different inputs to verify how the performance is related to the input size. We also measured the scalability either increasing the input size or the number of nodes, etc. In the remaining of this section, we report a detailed description of the experiments we conducted.

This section is organized as follows. We start off by providing a description of the prototype we have developed. Then, we give an overview of the experimental parameters. Finally, we group our results and discuss them in Sections 6.3–7.

6.1. Implementation

We have used the Hadoop 0.20.2 [3] framework, an open-source Java implementation of MapReduce, to implement WebPIE. Hadoop is designed for clusters of commodity machines and it can scale to cluster of thousand of machines. It uses a distributed file system built from the local disks of the participating machines, and manages execution details such as data transfer, job scheduling, and error management. The code used for our experiments is publicly available along with documentation and a tutorial [8].

6.2. Experimental parameters

The experimental parameters of our evaluation are described in this section. Table 6 shows an overview of these parameters, their range and the default values used. Unless otherwise specified, our experiments have been carried out using the default values. In the rest of this subsection, we briefly describe the meaning of these parameters.

Platform (*Default* value: DAS-4). We have performed most of our experiments on the DAS-4 VU cluster. In this cluster, each node is equipped with two quad-core Xeon processors, 24 GB of main memory and a single 2TB hard disk. The nodes are interconnected through Gigabit Ethernet.

We have also launched some experiments on the Amazon EC2 Cloud in order to facilitate comparison with future systems and to demonstrate the performance of WebPIE in the cloud. In this case, the Hadoop cluster consisted of 4 large Amazon instances, each with 7.5 GB of main memory, 850GB of hard disk space and 4 EC2 CUs, roughly equivalent to 2 cores.

Datasets (*Default* value: LDSR). We launched WebPIE on a set of real-world datasets: The Linked Data Semantic Repository (LDSR) dataset [6] (later renamed to FactForge [2]) consists of several commonly-used datasets like DBPedia, Freebase and Geonames. The Linked Life Data (LLD) dataset [7] is a curated collection of datasets from the biomedical domain. In the same domain, the Bio2RDF dataset [1] is currently the largest dataset with Semantic Web data, consisting of more than 24 billion triples. Finally, we have also used the Lehigh University Benchmark (LUBM) [13], which is a benchmark for RDF, consisting of generated information in the academic domain. LUBM can generate arbitrarily large datasets keeping with reasoning complexity. In Table 7, we report the number of statements, the number of statements for the OWL closure, and the number of distinct terms for each of these datasets. Also, for every dataset we report the supported reasoning complexity starting from one '+' if only limited RDFS/OWL reasoning is possible, and going to three '+' when almost all RDFS and OWL rules are supported. An intermediate number of '+' is to indicate the number of rules and should be used to get a rough indication of the reasoning complexity on that dataset.

Our system operates on compressed data, using the method in [32].

Reasoning complexity (*Default* value: OWL). We have performed experiments using both the RDFS ruleset (where the optimizations described in Section 3 apply) and the more expressive OWL-Horst ruleset (where the optimizations in Sections 3 and 4 apply). We have evaluated most of our optimizations using the OWL-Horst ruleset.

Table 6
Experimental parameters.

Parameter	Range	Default value
Platform	DAS-4, Amazon EC2	DAS-4
Dataset	LDSR, LLD, Bio2RDF, LUBM	LDSR
Reasoning complexity	RDFS, OWL	OWL
No. nodes	8, 16, 32, 64	32
Input size	1–100 billion	1 billion
RES	Fixed, Greedy	Fixed
DES	Always, End, Threshold	Always

Table 7
Datasets used in our experiments (size, size under OWL closure and number of distinct terms).

Dataset	Reasoning complexity	#Triples (Millions)	#Triples-OWL (Millions)	#Terms (Millions)
LDSR	+++	862	1790	259
LLD	++	694	1024	448
Bio2RDF	+	24000	25000	7302
LUBM	++	Flexible	Flexible	Flexible

Table 8
Reasoning time and throughput per dataset.

Dataset	Input size	RDFS		OWL	
		Throughput (millions)	Runtime (s)	Throughput (Ktps)	Runtime (s)
LDSR	862	1518	568	156	5512
LLD	694	2217	313	308	2253
Bio2RDF	24	4765	5122	656	37,221
LUBM	1101	1918	574	538	2046

Input size (*Default* value: 1.3B triples). Similarly, to measure the scalability of our approach regarding the input size we launched the reasoner varying the input size. We note that we cannot compare the performance of real-world datasets looking at their input size because they do not use the same language expressivity. For this reason, we will use the LUBM benchmark (which can scale arbitrarily in size) for evaluating the scalability of our system and the more expressive and realistic LDSR dataset in most other cases.

Execution strategies (*Default* value: fixed and always). In Section 4.5, we have presented different strategies for executing rules and eliminating duplicates. We will evaluate their impact on the total runtime for LDSR and LUBM.

6.3. Dataset and reasoning complexity

We launched WebPIE on different datasets and we report the runtimes in Table 8 for RDFS and OWL materialization. We make the following observations:

- RDFS reasoning is significantly faster than OWL reasoning, with a factor 5–10. This outcome was expected because RDFS has less and simpler rules than the OWL fragment and therefore it is easier to compute.
- The runtime and throughput of WebPIE is highly dependent on the logics employed by the input data. Moreover, the computation time for the two logics is not strongly correlated: if we consider the RDFS fragment, we note that LLD yields the best results. However, for the OWL fragment, then LUBM is the one with the best performance.
- For larger datasets, our system yields better throughput (e.g. for Bio2RDF we have at least two times higher throughput compared to the other datasets) because the framework overhead becomes much less relevant in the overall computation.
- For Bio2RDF, the ratio of derivations compared to the size of the input is low. This attests to the relatively weak inference possible over such large corpora.

To better illustrate how the input complexity affects the performance, we report in Fig. 5 the runtime of every MapReduce job when WebPIE is launched on LUBM and LDSR. We observe the following: (i) reasoning with the LDSR dataset is significantly more complex. Namely, while we need a total of 23 jobs for LUBM, while for LDSR, 53 jobs are required. (ii) The fastest jobs take around 60 s to finish. This is attributed to the platform overhead for starting a job. In practice, this means that the execution time is dominated by the platform overhead for starting jobs.

6.4. Scalability

We have evaluated the scalability of our approach in terms of input size and number of nodes. For these experiments, we have used the LUBM dataset, since it can be scaled to arbitrary size without any effect on its complexity.

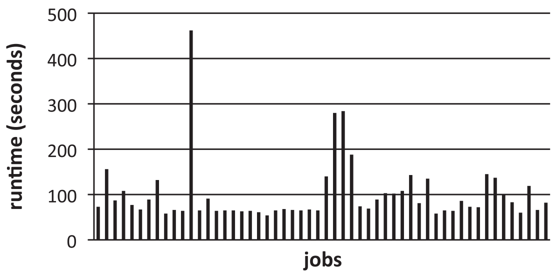
Fig. 6 summarizes our findings concerning the performance of our system for a varying number of nodes. Fig. 6(a) shows the runtime for calculating the closure of LUBM together with the (theoretical) linear speed–up and Fig. 6(b) shows the scaled speed-up. The latter is defined as $speed - up/no.nodes$, and is an indicator for the efficiency of our system, given additional computational nodes. A scaled speed-up of 1 means that given twice the number of nodes the system will perform twice as fast. We note that, for 16 nodes or less, our system has *superlinear* speed-up, meaning that for double the number of nodes, our system performs *better* than twice as fast. This is attributed to the fact that, with a larger number of nodes, we can store a larger part of the input in memory, which dramatically increases performance.

In Fig. 6(b), we also observe that, for 32 nodes, the scaled speed-up decreases drastically. This is because of the time required by the platform to start a job. Regardless of the number of nodes, calculating the closure of LUBM will always take more than approximately 25 min (a limitation imposed by the Hadoop platform).

Fig. 7 shows the runtime for an input size up to 100 billion triples on 64 nodes. Here, we see a similar situation as before: for small data sizes our throughput is reduced, since the runtime is dominated by platform overhead. Starting from 5 billion triples, the throughput improves dramatically as the platform overhead is amortized over the longer runtime. For input larger than 20 billion triples, we benefit less from keeping large portions of the data in memory and the throughput slightly decreases.

In general we notice that the performance is approximatively linear with regard to the input size. This can be surprising considering that reasoning is a task with a computational complexity worse than linear. The reason behind the linear behavior is that LUBM is a benchmark that generates input with a reasoning complexity proportional to its size. That means that if we double the input size then also the computational complexity is doubled. Such feature of LUBM fits with the purpose of this experiment because in order to measure the scalability we do not want that other factors like input complexity might influence the performance.

We have also analyzed the nodes' computational load during the reasoning task in order to understand if there are notable unbalances. To this purpose, we launched the reasoning process



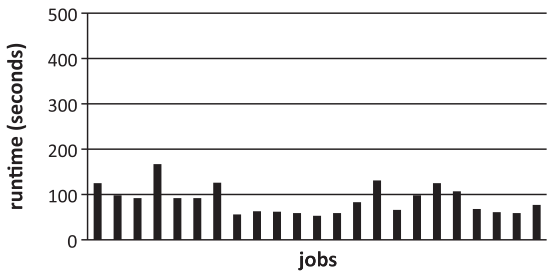


Fig. 5. Runtime per job for LDSR (left) and LUBM (right).

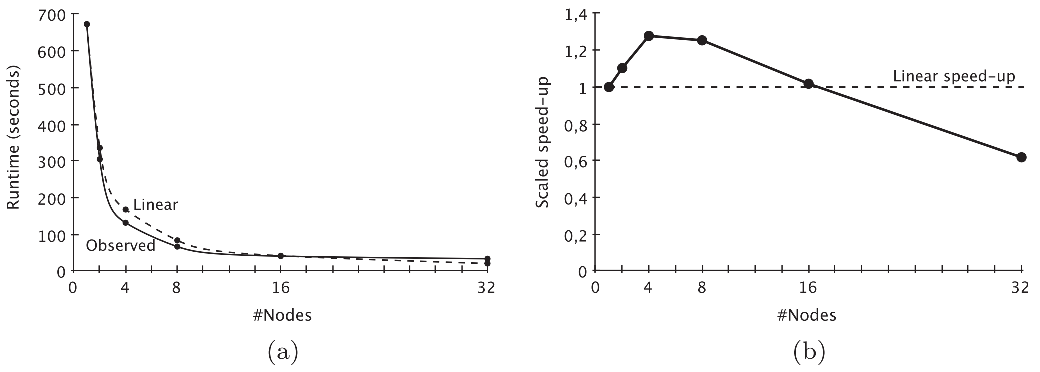
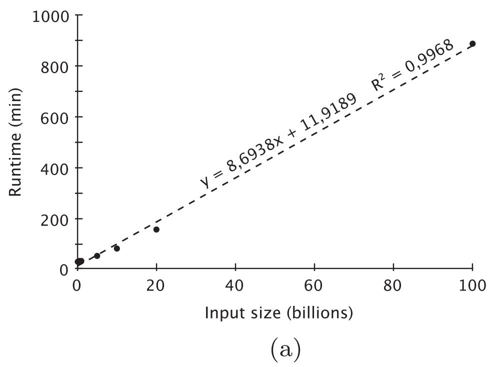


Fig. 6. (a) Runtime for the number of nodes (lower is better) and (b) scaled speed-up (higher is better).



# Triples (billions)	Runtime (min)	Throughput (Ktps)
0,1	29	58
0,5	32	262
1	35	481
5	52	1592
10	81	2047
20	156	2125
100	882	1889

(b)

Fig. 7. Runtime for increasing input size (on 64 nodes).

on the LUBM(8000) dataset using 32 DAS-4 machines. Since the reasoning MapReduce jobs are I/O bound, we focus on the amount of data processed by the nodes and we report the results in Fig. 8. In the first figure, we report the average amount of data that each node reads and writes during the map and reduce functions of the 23 MapReduce reasoning jobs. From the figure, we notice that the amount of intermediate data is considerably high on five cases. These jobs are the ones that clean the duplicates and such behavior is expected since they read and group the entire input in order to delete the duplicates. In the remaining cases, the amount of data read in input is higher than the amount effectively processed. This behavior indicates that in general only a small part of the input is relevant for reasoning.

In the second figure, we report the total amount of read and written data per node to verify that the computation is balanced across the nodes. Also in this case, we divided the total amount of data between the read and written in the map and reduce functions so that we can analyze their behavior independently. First we notice that the output of the map function is not equally distributed since there is a difference of a few gigabytes between the nodes. This behavior is explained as follows: in our prototype we split the triples in several files according to their predicate so that, depending on the reasoning rule, we can read a smaller amount of data skipping irrelevant data. For example, all the `rdf:type` triples are stored on files with extension ”type”. If there is a rule for which the ”type” triples cannot be used, we skipped reading them with consequent increase of performance. All the files are compressed and split in blocks of 64 megabytes. The blocks are distributed across the nodes and for each of them the node will apply the map function specific of the MapReduce job. Each time, the amount read is constant (64 megabytes) but the number of records might differ since the data is compressed and triples with the same predicate will take less space (due to compression optimizations). Therefore, some mappers might have to process more data and this explains the unbalance among them. We must note that this unbalance is not excessive since writing a few gigabytes of data does not take more than few minutes on a computation that is often more than a hour.

The amount of data processed during the reduce phase is uniformly spread across the nodes. This indicates that the optimizations that we explained in the previous sections do result in a good balancing with consequent benefit in terms of scalability.

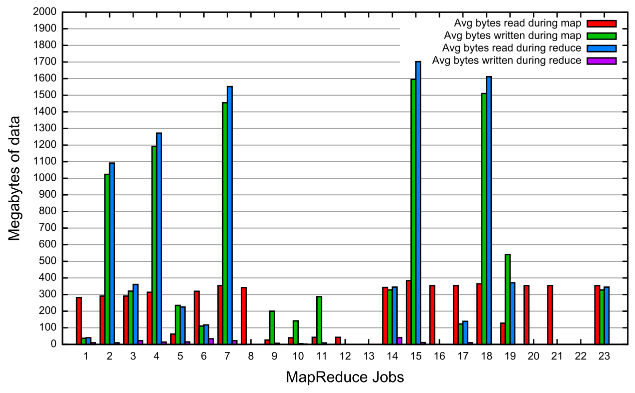
From the above, it is safe to conclude that our approach scales gracefully both in terms of the number of nodes and data size, given than the reasoning task is large enough.

6.5. Execution strategies

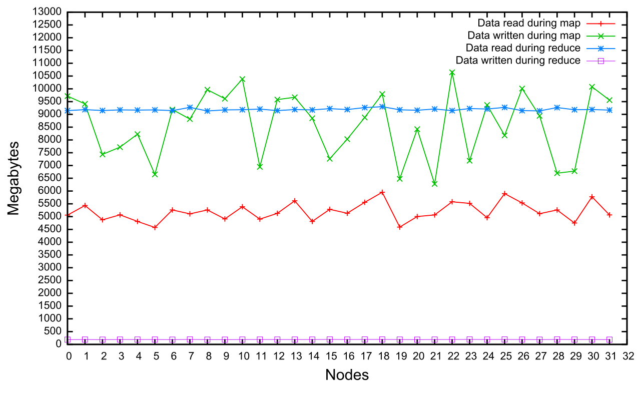
In Section 5, we have proposed a set of strategies based on some heuristics to limit the number of rule executions required to compute the closure.

We have launched all the combinations of rules and duplicates strategies on two datasets (LDSR and LUBM) and we report our findings in Table 9. We observe that applying different execution strategies indeed has a considerable effect on the performance in a range of approximately 10%. For LDSR, the best combination is the RES *fixed* with the DES *threshold* while for the LUBM dataset the best combination is the RES *greedy* with the DES *always*.

Considering the above, there is no clear pattern so as to which setting performs best. The gain derived by choosing a different strategy is never above 10%. There appears to be an interdependence between the strategies for rule execution and duplicate elimination and the complexity of the dataset (let us repeat that LUBM requires 23 jobs while LDSR requires 54). This is to be expected, since these optimizations aim at reducing the number of jobs and intermediate data processed, both of which are highly dependent on the input. A more complete analysis of the effect of changing rules strategy on different inputs goes beyond the



(a) Average r/w data per node



(b) Total r/w data per node

Fig. 8. Analysis of node computation per node during reasoning task.

scope of this work. Here we will just acknowledge a limited impact on the performance and a significant interdependence between the strategy and the particular input.

6.6. Platform

In other results (not shown), we have measured that, for the DAS-4, the jobs are I/O bound. Thus, our method would benefit from a hardware setup with higher I/O throughput at the expense of lower CPU power. On the other hand, for the Amazon EC2 Cloud, our jobs are mostly CPU bound. To facilitate comparison on a standard platform, we report that the execution time for LUBM (1 billion) on the four large instances on Amazon EC2 was 1064 minutes, 8 times slower than using 4 nodes on the DAS-4. This degrade in performance is expected, considering that our experiments were running on virtualized and lower-spec hardware.

6.7. Incremental reasoning

To evaluate our algorithms for incremental reasoning, we have used the LLD datasets that consists of several subdatasets and split it in several chunks. We initially performed reasoning on the first chunk, then added the second chunk, relaunched reasoning, and so on.

At first we have split the LLD datasets in four random chunks of roughly the same size. The performance is reported in Fig. 9(a). Next, to provide a more realistic scenario, we have chosen another criteria to split the dataset. This time, instead of having four random chunks, we have grouped its constituent datasets in four parts so that each increment contains data from new sources. The results are reported in Fig. 9(b).

What we expect is a non-linear reasoning behavior, where the reasoning time does not depend only on the size of the chunk but also on its complexity. To support this claim, suppose as example that we add a number of triples which have no connection with the existing ones. Because these triples have a limited impact, after reading the data the reasoner will stop in a relatively short time since nothing or only few will be derived. However, if we add one single triple which redefines a popular part of the schema, then the reasoner will have to recompute much of the closure and the reasoning time will be much higher.

The results confirm our hypothesis. In fact, we observe in the second table that the reasoning time is relatively low when we include datasets that do not yield new derivations, regardless of the size of the update. In this case the reasoner filters out all the old derivations as soon as possible, preventing the reasoner from recalculating anything that was already derived. However, because each MapReduce job takes some time to start, the execution time has a lower bound determined by the framework overhead for starting and finishing a job.

On the other side, we observe that the time to load the last increment is comparable to restarting the inference process. Again, this was expected because the last update has a high impact on the rest of the input. In this case, the reasoner has to launch a considerable number of jobs and the benefit of having computed part of the closure beforehand fades away when compared to the size of the computation.

Table 9
Rules execution strategies (RES) and duplicates elimination strategies (DES) on LDSR and LUBM.

(a) LDSR

RES	DES		
	always	*threshold*	*end*
fixed	5512	5063	5193
greedy	5084	5444	5247

(b) LUBM

RES	DES		
	always	*threshold*	*end*
fixed	2046	2388	2070
greedy	1999	2223	2246

Chunks	#Triples (millions)	Runtime (sec.)	Derivation (millions)
Random	173	1214	29
Random	172	1787	75
Random	169	1799	117
Random	167	2322	121

(a) Dataset split in four random chunks

Chunks	#Triples (millions)	Runtime (sec.)	Derivation (millions)
1. Uniprot	464	1231	273
2. Pubmed	201	481	0
3. Others	17	483	0
4. Others	13	2398	57

(b) Dataset split in four subdatasets

Fig. 9. Runtime for increasing input size (on 64 nodes).

If we split the dataset in random chunks, the reasoning complexity is spread among them. This means that every chunk will have a considerable impact. In this case the reasoner suffers the additional complexity brought from each chunk and the performance decreases even if the chunks have the same size, since every time it must perform reasoning also on the already existing data.

As a summary of this, we can conclude that incremental reasoning has performance which varies consistently depending on the input. If we add data, then the reasoning will be quick. However, if we add data with a significant impact then the performance decays and it becomes more convenient to rerun the reasoning on the entire input.

7. Related work

In this section we describe the related work regarding scalable reasoning with high performance.

Hogan et al. [15,16] compute the closure of an RDF graph using two passes over the data on a single machine. Initially they have implemented only a fragment of the OWL Horst semantics to allow more efficient materialization and to prevent ‘‘ontology hijacking’’. Later, the authors have extended their original approach to support a subset of OWL 2 RL and present an evaluation on 1.1 billion triples of a distributed implementation of their algorithms.

Schlicht and Stuckenschmidt [26] show peer-to-peer reasoning for the expressive ALC logic but focus on distribution rather than performance.

Soma and Prasanna [27] 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 [21] the authors present an inference engine that supports OWL 2 RL implemented inside the Oracle database. They describe a set of techniques to efficiently handle particular rules and they present an evaluation on a limited number of machines on datasets up to 13 billion triples. While they directly compare their performance against our work, we must point out that the two systems are quite different both in terms of hardware and software and therefore a direct comparison should be carefully weighted.

In [25,22], 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 costs. Experimental results were however only reported for datasets of up to 200M triples.

In [33], 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.

In [23], a method for distributed reasoning with EL++ using MapReduce is presented, which is applicable to the EL fragment of OWL 2. This work was inspired by ours and tackles a more complicated logic than the one presented here. It is still an on-going work and no experimental results are provided.

Newman et al. [24] 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). Husain et al. [19] report results for SPARQL querying using MapReduce for datasets up to 1.1 billion triples.

The work on Signal/Collect [28], introduces a new programming paradigm, targeted at handling graph data in a distributed manner. Although very promising, it is not comparable to our approach, since current experiments deal with much smaller graphs and are performed on a single machine.

Several proposed techniques are based on deterministic rendezvous-peers on top of distributed hashtables [10,12,20,9]. However, because of load-balancing problems due to the data distributions, these approaches do not scale [22].

Some Semantic Web stores support reasoning and scale to tens of billions of triples [4]. We have shown inference on a triple set which is one order of magnitude *larger* than reported anywhere (100 billion triples against 12 billion triples). Furthermore, our inference is 240 times *faster* (12 billion triples in 290 h for LUBM against 10 billion triples in 1.35 h on 64 nodes) against the best performing reasoner (BigOWLIM). For LDSR, BigOWLIM 3.1 processes 14 Ktriples per second [5] while our system yields a throughput of 156 Ktriples, on 32 nodes). It should be noted that the comparison of our system with RDF stores is not always meaningful, as our system does not support querying.

8. Conclusion

8.1. Summary

In this paper, we have shown a massively scalable technique for parallel OWL Horst forward inference and demonstrated inference over 100 billion triples.

In order to improve the performance, we have introduced a number of key optimizations to handle a specific set of rules. These optimizations are:

- Load the schema triples in memory and, when possible, execute the join on-the-fly instead of in the reduce phase.
- Perform the joins during the reduce phase and use the map function to group the triples in order to avoid duplicates.
- Execute the RDFS rules in a specific order to minimize the number of MapReduce jobs.
- Limit duplicates when performing joins between instance triples using contextual information.
- Limit the exponential derivation of owl:sameAs triples building a sameAs table.
- Perform redundant joins to avoid load balancing problems.

In addition, we have implemented mechanisms to apply rules with different strategies and added support to perform reasoning on incremental data.

Both in terms of processing throughput and maximum data size, our technique outperforms published approaches by a large margin.

8.2. 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 [17]; and (d) all the input is available locally in the distributed filesystem. The difference in performance on LLD, LDSR and LUBM shows that the complexity of the input data strongly affects performance. Although it is easy to create artificial datasets which would degrade 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.

8.3. Discussion on MapReduce

In this work, we have implemented a reasoning method using the MapReduce programming paradigm. In principle, the optimizations we have developed are not MapReduce-specific and can be applied on top of any infrastructure with similar workings. Originally, MapReduce operates in two discrete phases: a Map phase, where data is partitioned and a Reduce phase, where data in a given partition is processed. In some of our optimizations, we are deviating from this model by performing joins during the Map phase (effectively skipping the partitioning phase and operating on arbitrary parts of the input, together with a part of the input, replicated across all nodes).

On a larger scope, the Map and Reduce phases correspond to what a term partitioning system would do. In this sense, our system is similar to other systems doing term-partitioning (for example DHT-based systems). Replicating part of the input and performing a join in the Map phase would correspond to a broadcast, in such a context. Thus, we expect that many of the optimizations presented in this paper can be carrier over to any system doing term partitioning. Our choice of MapReduce was mainly made for reasons of performance and scalability.

8.4. 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 choose 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.

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.

Acknowledgements

We would like to thank Amazon Inc. for providing us with credits to run experiments on the EC2 cloud. This work was partially funded by the EU-IST project LarKC (FP7-215535).

Appendix A. MapReduce reasoning algorithms

In the appendix, we report on the MapReduce algorithms corresponding to the optimizations discussed in Sections 3 and 4. In Appendices A.1 and A.2 we report on the algorithms for RDFS and OWL reasoning respectively. For clarity purposes, we do not describe the algorithms in details but more on a higher level, using pseudo code instead of a real language.

The feature of incremental updates slightly changes the reasoning algorithms because it requires that the map and reduce functions do an additional check to see whether the timestamps of the input triples meet the requirements to allow a new derivation. However, in order to keep the algorithms readable and concise, we add this feature only to the Algorithm 4, since this operation is completely analogous in the other cases.

A.1. RDFS MapReduce algorithms

We grouped the RDFS rules in four MapReduce jobs, to which we will refer to as SUBPROP, DOMAINRANGE, SUBCLASS and SPECIAL_CASES. These jobs are executed in sequence as described in Algorithm 3. The last job is not always executed because it refers to a special case that rarely occurs.

In the remaining of this section we discuss each of these reasoning algorithms.

Algorithm 3. RDFS overall algorithm

```
rdfs_reasoning(data) {
  apply_job(data, SUBPROP);
  derived+=apply_job(data + derived, DOMAINRANGE);
  derived=clean_duplicates(data, derived);
  derived+=apply_job(data + derived, SUBCLASS);
  if (derived_special_cases(derived) == true)
    derived+=apply_job(data + derived, SPECIAL_CASES);
}
```

SUBPROP applies rules 5 and 7, which concern sub-properties, and is reported in Algorithm 4. Since the schema triples are loaded in memory, these rules can be applied simultaneously. To avoid generation of duplicates, we follow the principle of setting as the tuple's key the triple's parts that are used in the derivation. This is possible because all inferences are drawn on an instance triple and a schema triple and we load all schema triples in memory. That means that for rule 5, we output as key the triple's subject while for rule 7 we output a key consisting of the subject and object. We add an initial flag to keep the groups separated since, later, we have to apply a different logic that depends on the rule. In case we apply rule 5, we output the triple's object as value, otherwise we output the predicate. Also, the map function checks whether the highest timestamp between the instance and the schema triples is higher than the last execution to prevent the derivation of information that was already derived.

The reducer reads the flag of the group's key and applies to corresponding rule. In both cases, it first filters out the duplicates in the values. Then, it recursively matches the tuple's values against the schema and saves the output in a set. Once the reducer has finished with this operation, it outputs the new triples using the information in the key and in the derivation output set.

DOMAINRANGE applies rules 2 and 3, as shown in Algorithm 5. We use a similar technique as before to avoid generating duplicates. In this case, we emit as key the triple's subject and as value the predicate. We also add a flag so that the reducers know whether they have to match it against the domain or range schema. Pairs about domain and range will be grouped together if they share the same subject since the two rules might derive the same triple.

SUBCLASS applies rules 9, 11, 12, and 13, which are concerned with sub-class relations. The procedure, shown in Algorithm 6, is similar to the previous job with the following difference: during the map phase, we do not filter the triples which match with the schema but forward everything to the reducers instead. In doing so, this job also eliminates the duplicates against the input and we do not need to launch an additional job after this.

SPECIAL_CASES refers to the special cases when rules 12 and 13 derive information which might fire rules 5, 7, 9 and 11. During the map phase, it loads in memory the subproperties of *rdfs:member* and the subclasses of *rdfs:Literal* and it performs the join between the triples in input with this schema. During the reduce, the job materializes the results of this join. Since this algorithm does not introduce any new challenges, we do not report the pseudocode.

Algorithm 4. RDFS sub-property reasoning (SUBPROP)

```
map(key, value):
  if (subproperties.contains(value.predicate) && // for rule 7
      max(value.timestamp,
   subproperties.get(value.predicate).timestamp) >
last_execution_timestamp)
          key = "1" + value.subject + "-" + value.object
      emit(key, value.predicate)
  if (subproperties.contains(value.object) && // for rule 5
      value.predicate == "rdfs:subPropertyOf" &&
      max(value.timestamp,
   subproperties.get(value.predicate).timestamp) >
          last_execution_timestamp)
    key = "2" + value.subject
    emit(key, value.object)
reduce(key, values):
  values = values.unique // filter duplicate values
  switch (key[0])
    case 1: // we are doing rule 7: subproperty inheritance
      for (predicate in values)
        // iterate over the predicates emitted in the map and
   collect superproperties
      superproperties.add(subproperties.recursive_
   get(value))
    for (superproperty in superproperties)
      // iterate over superproperties and emit instance triples
      emit(null, triple(key.subject, superproperty, key.object)
    case 2: // we are doing rule 5: subproperty transitivity
    for (predicate in values)
      // iterate over the predicates emitted in the map, and
   collect superproperties
      superproperties.add(subproperties.recursive_
   get(value))
    for (superproperty in superproperties)
      // emit transitive subproperties
      emit(null, triple(key.subject, "rdfs:subPropertyOf",
   superproperty))
```

Algorithm 5. RDFS domain and range reasoning (DOMAINRANGE)

```
map(key, value):
  if (domains.contains(value.predicate)) then // for rule 2
    key = value.subject
    emit(key, value.predicate + "d")
  if (ranges.contains(value.predicate)) then // for rule 3
    key = value.object
    emit(key, value.predicate +"r")
reduce(key, values):
  values = values.unique // filter duplicate values
  for (predicate in values)
    switch (predicate.flag)
      case "r": // rule 3: find the range for this predicate
        types.add(ranges.get(predicate))
      case "d": // rule 2: find the domain for this predicate
        types.add(domains.get(predicate))
  for (type in types)
    emit(null, triple(key, "rdf:type", type))
```

Algorithm 6. RDFS sub-class reasoning (SUBCLASS)

```
map(key, value):
  if (value.predicate = "rdf:type")
    key = "0" + value.predicate
    emit(key, value.object)
  if (value.predicate = "rdfs:subClassOf")
    key = "1" + value.predicate
    emit(key, value.object)
reduce(key, values):
  values = values.unique // filter duplicate values
  for (class in values)
    superclasses.add(subclasses.get_recursively(class))
  switch (key[0])
    case 0: // we are doing rdf:type
      for (class in superclasses)
        if !values.contains(class)
          emit(null, triple(key.subject, "rdf:type", class))
    case 1: // we're doing subClassOf
      for (class in superclasses)
        if !values.contains(class)
          emit(null, triple(key.subject, "rdfs:subClassOf",
   class))
```

Algorithm 7. Overall OWL reasoning algorithm

```
owl_readoning(data):
  boolean first_time=true;
  while (true) {
    derived=rdfs_reasoning(data);
    if (derived == null && first_time == false)
      return data;
    data= data + derived;
    do {// Do fixpoint iteration for ter Horst rules
      derived=apply_job(data, NOT_RECURSIVE_JOB);
      derived+=apply_recursevily_job(data + derived,
   TRANSITIVITY_JOB);
      derived=clean_duplicates(data, derived);
      derived+=apply_recursevily_job(data + derived,
   SAME_AS_TRANSITIVITY_JOB);
      derived=clean_duplicates(data, derived);
      derived+=apply_job(data + derived,
   SAME_AS_INHERIT_1_JOB);
      derived+=apply_job(data + derived,
   SAME_AS_INHERIT_2_JOB);
      derived+=apply_job(data + derived,
   SAME_AS_INHERIT_3_JOB);
      derived=clean_duplicates(data, derived);
      derived+=apply_job(data + derived,
   EQUIVALENCE_JOB);
      derived+=apply_job(data + derived, HAS_VALUES_JOB);
      derived=clean_duplicates(data, derived);
      derived+=apply_job(data + derived,
   SOME_ALL_VALUES_JOB);
      derived=clean_duplicates(data, derived);
      data= data + derived;}
    while (derived != null);
    first_time=false;
  }
```

A.2. OWL MapReduce algorithms

OWL reasoning requires launching jobs in a loop until the rules stop deriving any conclusion. In Algorithm 7, we report the overall reasoning algorithm. It consists of a main loop where it first executes all the RDFS rules (see the previous section) and then executes the OWL rules until all rules do not derive anything anymore. This algorithm implements the strategy *fixed*, where the rules are executed in a fixed order. For the other strategy, the order of the rules is dynamically rearranged depending on the amount of information that they have previously derived.

The OWL rules are implemented in ten MapReduce jobs. We describe each of them in the remainder of this section.

NOT_RECURSIVE_JOB executes rules 1, 2, 3, and 8 and is reported in Algorithm 8. These rules can be grouped and executed together. Their execution exploits the optimizations presented for the RDFS fragment. The instance triples are filtered during the map phase using the schema loaded in memory. The partitioning is done to avoid duplicates and the reduce function materializes the new triples.

TRANSITIVITY_JOB executes rule 4 and is reported in Algorithm 9. 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.

SAME_AS_TRANSITIVITY_JOB executes the logic of rule 7 to build the *sameAs* table and is reported in Algorithm 10. The sameAs triples are 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.

SAME_AS_INHERIT_*_JOBS is a sequence of three jobs that encode the logic of rule 11 and replace in the input triples every resource that appear in the *sameAs* table with its corresponding canonical representation. The first job counts the resources and identifies the most popular one. This is done to prevent load balancing issues in the following jobs. The second and third jobs perform the replacements. The replacements are done by deconstructing the statements and executing the join between the *sameAs* table and the input triples in the classical way. We do not report the pseudocode of these jobs because the technique is completely analogous to the data decompression technique described in [32] with the only difference being that here we perform the join against the *sameAs* table and not the dictionary.

EQUIVALENCE_JOB executes rules 12 and 13. The algorithm is not reported because its implementation is straightforward. These rules work with schema triples and therefore do not present any particular challenges. The corresponding job loads the schema in memory and derives the information without any duplicate derivations using the RDFS optimizations.

HAS_VALUE_JOB executes rules 14. Similarly, this algorithm performs the join between the schema triples in the nodes' main memory and filters the instance triples during the map phase. The reduce phase simply materializes the derivation while eliminating the eventual duplicates between the derivation.

SOME_ALL_VALUES_JOB executes rules 15 and 16 and is reported in Algorithm 11. 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. Then, 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, we have all possible bindings for *x* and *p*. Thus, we can partition on `xv` and perform the join during the reduce phase. The algorithm for `owl:allValuesFrom` is analogous to this one and not reported for conciseness.

Algorithm 8. OWL not recursive rules (`NOT_RECURSIVE_JOB`)

```
map(key, triple):
  if (functional_properties.contains(triple.predicate)) then //Rule 1
    emit('F',triple.subject,triple.predicate, triple.object);
  if (inverse_functional_properties.contains(triple.predicate)) then //Rule 2
    emit('F',triple.object,triple.predicate, triple.subject);
  if (symmetric_properties.contains(triple.predicate)) then //Rule 3
    emit('S',triple.subject,triple.object, triple.predicate);
  if (inverse_properties.contains(triple.predicate)) then //Rule 8
    emit('I',triple.subject,triple.object, triple.predicate);
reduce(key, values):
    switch (key[0])
      case 'F' : for (value in values) do //Process rule 1 and 2
          synonyms.add(value)
        for (synonym in synonyms) do
          emit('synonyms.min_value, owl:sameAs, synonym);
    case 'S' : for (value in values) do //Process rule 3
          emit(key.object, value, key.subject);
    case 'I' : for (value in values) do //Process rule 8
        inverse_property = inverse_properties.get(value)
          emit(key.object, inverse_property, key.subject);
```

Algorithm 9. OWL transitivity closure, *p* rule 4 (`TRANSITIVITY_JOB`)

```
map(key, triple):
  n = job_config.get_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));
```

Algorithm 10. OWL build sameAs table, *p* rule 7 (SAME_AS_TRANSITIVITY_JOB)

```
map(key, edge):
  emit(edge.from, forward, edge.to);
  emit(edge.to, backward, edge.from);
reduce(key, values):
  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);
```

Algorithm 11. OWL someValuesFrom and allValuesFrom reasoning, *p* rule 15 and 16 (SOME_ALL_VALUES_JOB)

```
map(key, triple):
  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, values):
  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));
```

References

[1] Bio2RDF, <http://bio2rdf.org>, 2010.
[2] FactForge, <http://www.factforge.com>, 2010.
[3] Hadoop, <http://hadoop.apache.org>, 2010.
[4] Large triple stores wiki pagelarge triple stores wiki page, <http://esw.w3.org/topic/LargeTripleStores>, 2010.
[5] LarKC deliverable 5.2.2, <http://hadoop.apache.org>, 2010.
[6] Linked Data Semantic Repository (LDSR), <http://www.ontotext.com/ldsr/>, 2010.
[7] Linked Life Data (LLD), <http://linkedlifedata.com>, 2010.
[8] WebPIE website, <http://www.cs.vu.nl/webpie>, 2010.
[9] D. Battré, A.Höing, F. Heine, 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.
[10] M. Cai, M. Frank, RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network, in: Proceedings of the International World-Wide Web Conference. 2004.
[11] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI), 2004, pp. 137–147.
[12] Q. Fang, Y. Zhao, G. Yang, W. Zheng, Scalable distributed ontology reasoning using DHT-based partitioning, in: Proceedings of the Asian Semantic Web Conference (ASWC), 2008.
[13] Y. Guo, Z. Pan, J. Heflin, LUBM: a benchmark for OWL knowledge base systems, Journal of Web Semantics 3 (2005) 158–182.
[14] P. Hayes (Ed.), RDF Semantics, W3C Recommendation, 2004.
[15] A. Hogan, A. Harth, A. Polleres, Scalable authoritative OWL reasoning for the web, International Journal on Semantic Web and Information Systems 5 (2) (2009).
[16] A. Hogan, J. Pan, A. Polleres, S. Decker, Saor: Template rule optimisations for distributed reasoning over 1 billion linked data triples, The Semantic Web - ISWC 2010,2010,pp.337–353.
[17] A. Hogan, A. Polleres, A. Harth, Saor: authoritative reasoning for the web, in: Proceedings of the Asian Semantic Web Conference (ASWC), 2008.
[18] 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) (2005) 79–115.
[19] M.F. Husain, P. Doshi, L. Khan, B. Thuraisingham, Storage and retrieval of large rdf graph using hadoop and mapreduce, in: M.G. Jaatun, G. Zhao, C. Rong (Eds.), Cloud Computing, vol. 5931, Springer, Berlin, Heidelberg, 2009, pp. 680–686 (Chapter 72).
[20] Z. Kaoudi, I. Miliaraki, M. Koubarakis, RDFS reasoning and query answering on top of DHTs, in: Proceedings of the International Semantic Web Conference (ISWC), 2008.
[21] V. Kolovski, Z. Wu, G. Eadon, Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system, in: The Semantic Web – ISWC 2010, 2010, pp. 436–452.
[22] S. Kotoulas, E. Oren, F. van Harmelen, Mind the data skew: distributed inferencing by speeddating in elastic regions, in: Proceedings of the WWW, 2010.
[23] R. Mutharaju, F. Maier, P. Hitzler, A mapreduce algorithm for el+, in: Proceedings of the 23rd International Workshop on Description Logics (DL2010), Waterloo, Canada, 2010.
[24] A. Newman, Y. Li, J. Hunter, Scalable semantics the silver lining of cloud computing, in: Proceedings of the 4th IEEE International Conference on eScience, 2008.
[25] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, et al., Marvin: distributed reasoning over large-scale semantic web data, Journal of Web Semantics 7 (4) (2009) 305–316.
[26] A. Schlicht, H. Stuckenschmidt, Peer-to-peer reasoning for interlinked ontologies, International Journal of Semantic Computing (2010) (Special Issue on Web Scale Reasoning).
[27] R. Soma, V. Prasanna, Parallel inferencing for OWL knowledge bases, in: International Conference on Parallel Processing, 2008, pp. 75–82.
[28] P. Stutz, A. Bernstein, W. Cohen, Signal/collect: Graph algorithms for the (semantic) web, in: Proceedings of the ISWC, Shanghai, China, 2010.
[29] J. Urbani, S. Kotoulas, J. Maassen, N. Drost, et al., Webpie: a web-scale parallel inference engine, 2010 (1st prize at the 3rd IEEE SCALE challenge at CCGrid).
[30] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, et al., Owl reasoning with mapreduce: calculating the closure of 100 billion triples, in: Proceedings of the ESWC 2010.
[31] J. Urbani, S. Kotoulas, E. Oren, F. van Harmelen, Scalable distributed reasoning using mapreduce, in: Proceedings of the ISWC, 2009.
[32] J. Urbani, J. Maassen, H. Bal, Massive semantic web data compression with mapreduce, in: HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, New York, NY, USA, 2010, pp. 795–802.
[33] J. Weaver, J. Hendler, Parallel materialization of the finite rdfs closure for hundreds of millions of triples, in: 8th International Semantic Web Conference (ISWC2009), 2009.