Scalable Distributed Reasoning using MapReduce

Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen

Department of Computer Science, Vrije Universiteit Amsterdam, the Netherlands

Abstract. We address the problem of scalable distributed reasoning, proposing a technique for materialising the closure of an RDF graph based on MapReduce. We have implemented our approach on top of Hadoop and deployed it on a compute cluster of up to 64 commodity machines. We show that a naive implementation on top of MapReduce is straightforward but performs badly and we present several non-trivial optimisations. Our algorithm is scalable and allows us to compute the RDFS closure of 865M triples from the Web (producing 30B triples) in less than two hours, faster than any other published approach.

1 Introduction

In this paper, we address the problem of *scalable distributed reasoning*. Most existing reasoning approaches are centralised, exploiting recent hardware improvements and dedicated data structures to reason over large-scale data [8, 11, 17]. However, centralised approaches typically scale in only one dimension: they become faster with more powerful hardware.

Therefore, we are interested in parallel, distributed solutions that partition the problem across many compute nodes. Parallel implementations can scale in two dimensions, namely hardware performance of each node and the number of nodes in the system. Some techniques have been proposed for distributed reasoning, but, as far as we are aware, they do not scale to orders of 10^8 triples.

We present a technique for materialising the closure of an RDF graph in a distributed manner, on a cluster of commodity machines. Our approach is based on MapReduce [3] and it efficiently computes the closure under the RDFS semantics [6]. We have also extended it considering the OWL Horst semantics [9] but the implementation is not yet competitive and it is should be considered as future work. This paper can be seen as a response to the challenge posed in [12] to exploit the MapReduce framework for efficient large-scale Semantic Web reasoning.

This paper is structured as follows: we start, in Section 2, with a discussion of the current state-of-the-art, and position ourselves in relation to these approaches. We summarise the basics of MapReduce with some examples in Section 3. In Section 4 we provide an initial implementation of forward-chaining

RDFS materialisation with MapReduce. We call this implementation "naive" because it directly translates known distributed reasoning approaches into MapReduce. This implementation is easy to understand but performs poorly because of load-balancing problems and because of the need for fixpoint iteration. Therefore, in Section 5, an improved implementation is presented using several intermediate MapReduce functions. Finally, we evaluate our approach in Section 6, showing runtime and scalability over various datasets of increasing size, and speedup over increasing amounts of compute nodes.

2 Related work

Hogan *et al.* [7] compute the closure of an RDF graph using two passes over the data on a single machine. They implement only a fragment of the OWL Horst semantics, to allow efficient materialisation, and to prevent "ontology hijacking". Our approach borrows from their ideas, but by using well-defined MapReduce functions our approach allows straightforward distribution over many nodes, leading to improved results.

Mika and Tummarello [12] use MapReduce to answer SPARQL queries over large RDF graphs, and mention closure computation, but do not provide any details or results. In comparison, we provide algorithm details, make the code available open-source, and report on experiments of up to 865M triples.

MacCartney *et al.* [13] show that graph-partitioning techniques improve reasoning over first-order logic knowledge bases, but do not apply this in a distributed or large-scale context. Soma and Prasanna [15] present a technique for parallel OWL inferencing through data partitioning. Experimental results show good speedup but on relatively 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 previous work [14] we have presented a technique based on data-partitioning in a self-organising P2P network. A load-balanced auto-partitioning approach was used without upfront partitioning cost. Conventional reasoners are locally executed and the data is intelligently exchanged between the nodes. The basic principle is substantially different from the work here presented and experimental results were only reported for relatively small datasets of up to 15M triples.

Several techniques have been proposed based on deterministic rendezvous-peers on top of distributed hashtables [1, 2, 4, 10]. However, these approaches suffer of load-balancing problems due to the data distributions [14].

3 What is the MapReduce framework?

MapReduce is a framework for parallel and distributed processing of batch jobs [3] on a large number of compute nodes. 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 based on key/value pairs: the map function

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 amount 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, equalling occurrences
  emit(key, count);
```

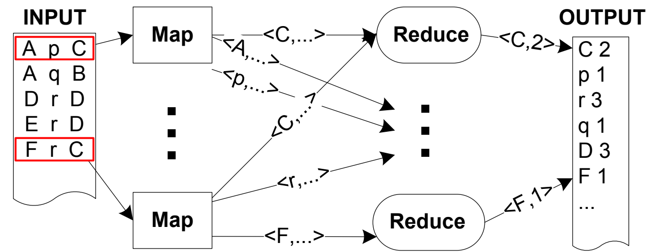


Fig. 1. MapReduce processing

processes a key/value pair and produces a set of new key/value pairs; the reduce merges all intermediate values with the same key into final results.

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 these the number of values into an aggregate term count (one value was emitted for each term occurrence).

This job could be executed as shown in Figure 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.

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,

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 q 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

Table 1. RDFS rules [6]

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.
– the *reduce* operates on all pieces of data that share some key, assigned in a *map*. 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.

4 Naive RDFS reasoning with MapReduce

The closure of an RDF input graph under the RDFS semantics [6] can be computed by applying all RDFS rules iteratively on the input until no new data is derived (fixpoint). The RDFS rules, shown in Table 1, have one or two antecedents. For brevity, we ignore the former (rules 1, 4a, 4b, 6, 8, 10, 12 and 13) since these can be evaluated at any point in time without a join. Rules with two antecedents are more challenging to implement since they require a join over two parts of the data.

4.1 Encoding an example RDFS rule in MapReduce

Applying the RDFS rules means performing a join over some terms in the input triples. Let us consider for example rule 9 from Table 1, which derives `rdf:type` based on the sub-class hierarchy. We can implement this join with a *map* and *reduce* function, as shown in Figure 2 and Algorithm 2:

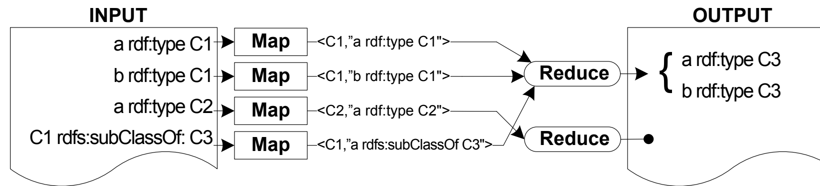


Fig. 2. Encoding RDFS rule 9 in MapReduce.

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));
```

In the *map*, 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 sub-class 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.

4.2 Complete RDFS reasoning: the need for fixpoint iteration

If we perform this *map* once (over all input data), and then the *reduce* once, we will not find *all* corresponding conclusions. For example, to compute the transitive closure of a chain of n `rdfs:subClassOf`-inclusions, we would need to iterate the above *map/reduce* steps n times.

Obviously, the above *map* and *reduce* functions encode only rule 9 of the RDFS rules. We would need to add other, similar, *map* and *reduce* functions 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, chain these functions to each other, and iterate these until we reach some fixpoint.

5 Efficient RDFS reasoning with MapReduce

The previously presented implementation is straightforward, but is inefficient because it produces duplicate triples (several rules generate the same conclusions) and because it requires fixpoint iteration. We encoded, as example, only rule 9 and we launched a simulation over the Falcon dataset, which contains 35 million triples. After 40 minutes the program had not yet terminated, but had already generated more than 50 billion triples. 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. Though the amount of duplicate triples depends on the specific data set, a valid approach should be able to efficiently deal with real world example like Falcon.

In the following subsections, we introduce three optimisations to greatly decrease the number of jobs and time required for closure computation:

5.1 Loading schema triples in memory

Typically, schema triples are far less numerous than instance triples [7]; As also shown in Table 2, our experimental data[1] indeed exhibit a low ratio between schema and instance triples. In combination with the fact that RDFS rules with two antecedents include at least one schema triple, we can infer that joins are made between a large set of instance triples and a small set of schema triples. For example, in rule 9 of Table 1 the set of `rdf:type` triples is typically far larger than the set of `rdfs:subClassOf` triples. As our first optimisation, we can load the small set of `rdfs:subClassOf` triples in memory and launch a MapReduce job that streams the instance triples and performs joins with the in-memory schema triples.

5.2 Data grouping to avoid duplicates

The join with the schema triples can be physically executed either during the map or during the reduce phase of the job. After initial experiments, we have concluded that it is faster to perform the join in the reduce, since doing so in the map results in producing large numbers of duplicate triples.

[1] from the Billion Triple challenge 2008, `http://www.cs.vu.nl/~pmika/swc/btc.html`

schema type	amount	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%

Table 2. Schema triples (amount and fraction of total triples) in datasets

Let us illustrate our case with an example based on rule 2 (`rdfs:domain`). Assume 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 since they will need to be stored locally and be transfered over the network.

We can avoid the generation of duplicates if we first group the triples by subject and then we execute the join over the single group. We can do it by designing a mapper that outputs an intermediate tuple that has as key the triple's subject and as value the predicate. In this way the triples will be grouped together and we will execute the join only once, avoiding generating duplicates.

In general, we set as key those parts of the input triples that are also used in the derived triple. The 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. Since the key is used to partition the data, for a given rule, all triples that produce some new triple will be sent to the same reducer. It is then trivial to output that triple only once in the reducer. As value, we emit those elements of the triple that will be matched against the schema.

5.3 Ordering the application of the RDFS rules

We analyse the RDFS ruleset with regard to input and output of each rule, to understand which rule may be triggered by which other rule. By ordering the execution of rules we can limit the number of iterations needed for full closure. As explained before, we ignore some of the rules with a single antecedent (1, 4, 6, 8, 10) without loss of generality: these can be implemented at any point in time without a join, using a single pass over the data. We first categorise 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 categorise the rules based on the predicates in their antecedents:

- rules 5 and 10 operate only on triples with sub-class or sub-property triples,
- rules 9, 12 and 13 operate on triples with type, sub-class, and sub-property,

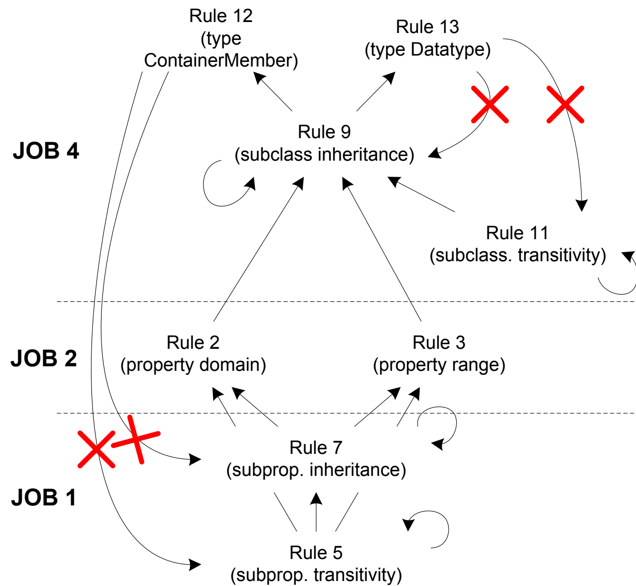


Fig. 3. Relation between the various RDFS rules

– rule 2, 3 and 7 can operate on arbitrary triples.

Figure 3 displays the relation between the RDFS rules, connecting rules based on their input and output (antecedents and consequents). An ideal execution should proceed from the bottom of the picture to the top: first apply the transitivity rules (rule 5 and 11), then apply rule 7, then rule 2 and 3, then rule 9 and finally rules 12 and 13.

It may seem that rule 12 and 13 could produce triples that would serve as input to rules 5 and 11; however, looking carefully we see that this is not the case: Rule 12 outputs `(?s rdfs:subPropertyOf rdfs:member)`, rule 13 outputs `(?s rdfs:subClassOf rdfs:Literal)`. For rules 5 and 11 to fire on these, *rdfs:member* and *rdfs:Literal* must have been be defined as sub-classes or sub-properties of something else. However, in RDFS none of these is a sub-class or sub-property of anything. They could of course be super-classed by arbitrary users on the Web. However, such "unauthorised" statements are dangerous because they can cause ontology hijacking and therefore we ignore them following the advice of [7]. Hence, the output of rules 12 and 13 cannot serve as input to rules 5 and 11. Similarly, rules 2 and 3 cannot fire.

Furthermore, rule 9 cannot fire after rule 13, since this would require using literals as subjects, which we ignore as being non-standard RDF. The only rules that could fire after rule 12 are rules 5 and 7. For complete RDFS inferencing, we would need to evaluate these rules for each container-membership property found in the data, but as we will show, in typical datasets these properties occur very rarely.

As our third optimisation, we conclude that instead of having to iterate over all RDFS rules until fixpoint, it is sufficient to process them only once, in the order indicated in Figure 3.

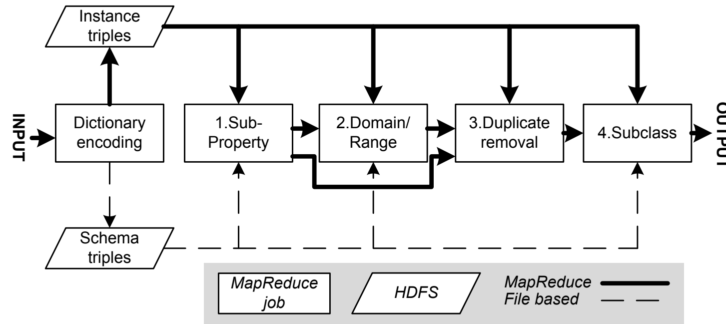


Fig. 4. Data flow. The solid lines refer to data split partitioned using MapReduce while the dashed lines refer to shared data.

5.4 The complete picture

In this section, we present an updated algorithm implementing the above optimisations. The complete algorithm consists of five sequential MapReduce jobs, as shown in Figure 4. First, we perform dictionary encoding and extract the schema triples to a shared distributed file system. Then, we launch the RDFS reasoner that consists in a sequence of four MapReduce jobs.

The first job applies the rules that involve the sub-property relations. The second applies the rules concerning domain and range. The third cleans up the duplicated statements produced in the first step and the last applies the rules that use the sub-class relations. In the following subsections, each of these jobs is explained in detail.

Distributed dictionary encoding in MapReduce To reduce the physical size of the input data, we perform a dictionary encoding, in which each triple term is rewritten into a unique and small identifier. We have developed a novel technique for distributed dictionary encoding using MapReduce, rewriting each term into an 8-byte identifier; the encoding scales linearly with the input data. Due to space limitations, we refer the reader to [16]. Encoding all 865M triples takes about 1 hour on 32 nodes. Note that schema triples are extracted here.

First job: apply rules on sub-properties The first job applies rules 5 and 7, which concern sub-properties, as shown in Algorithm 3. Since the schema triples are loaded in memory, these rules can be applied simultaneously.

Algorithm 3 RDFS sub-property reasoning

```
map(key, value):
  // key: null
  // value: triple
  if (subproperties.contains(value.predicate)) // for rule 7
    key = "1" + value.subject + "-" + value.object
    emit(key, value.predicate)
  if (subproperties.contains(value.object) &&
      value.predicate == "rdfs:subPropertyOf") // for rule 5
    key = "2" + value.subject
    emit(key, value.object)

reduce(key, iterator values):
  // key: flag + some triples terms (depends on the flag)
  // values: triples to be matched with the schema
  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))
```

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 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.

The reducer reads the flag of the group's key and applies to corresponding rule. In both cases, it first filters out 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.

This algorithm will not derive a triple more than once, but duplicates may still occur between the derived triples and the input triples. Thus, at a later stage, we will perform a separate duplicate removal job.

Second job: apply rules on domain and range The second job applies rules 2 and 3, as shown in Algorithm 4. Again, we use a similar technique to avoid generating duplicates. In this case, we emit as key the triple's subject and as

Algorithm 4 RDFS domain and range reasoning

```
map(key, value):
  // key: null
  // value: triple
  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, iterator values):
  // key: subject of the input triples
  // values: predicates to be matched with the schema
  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))
```

value the predicate. We also add a flag so that the reducers know if they have to match it against the domain or against the range schema. Tuples about domain and range will be grouped together if they share the same subject since the two rules might derive the same triple.

Third job: delete duplicate triples The third job is simpler and eliminates duplicates between the previous two jobs and the input data. Due to space limitations, we refer the reader to [16].

Fourth job: apply rules on sub-classes The last job applies rules 9, 11, 12, and 13, which are concerned with sub-class relations. The procedure, shown in Algorithm 5, is similar to the previous job with the following difference: during the map phase we do not filter the triples but forward everything to the reducers instead. In doing so, we are able to also eliminate the duplicates against the input.

6 Experimental results

We use the Hadoop[2] framework, an open-source Java implementation of MapReduce. Hadoop is designed to efficiently run and monitor MapReduce applications on clusters of commodity machines. It uses a distributed file system and manages execution details such as data transfer, job scheduling, and error management.

Our experiments were performed on the DAS-3 distributed supercomputer[3] using up to 64 compute nodes with 4 cores and 4GB of main memory each, using

[2] http://hadoop.apache.org
[3] http://www.cs.vu.nl/das3

Algorithm 5 RDFS sub-class reasoning

```
map(key, value):
  // key: source of the triple (irrelevant)
  // value: triple
  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, iterator values):
  //key: flag + triple.subject
  //iterator: list of classes
  values = values.unique // filter duplicate values

  for (class in values)
    superclasses.add(subclasses.get_recursively(class))

  switch (key[0])
   case 0: // we're 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))
```

dataset	input	output	time	σ
Wordnet	1.9M	4.9M	3'39"	9.1%
Falcon	32.5M	863.7M	4'19"	3.8%
Swoogle	78.8M	1.50B	7'15"	8.2%
DBpedia	150.1M	172.0M	5'20"	8.6%
others	601.5M			
all	864.8M	30.0B	56'57"	1.2%

Table 3. Closure computation using datasets of increasing size on 32 nodes

Gigabit Ethernet as an interconnect. We have experimented on real-world data from the Billion Triple Challenge 2008[4]. An overview of these datasets is shown in Table 3, where dataset *all* refers to all the challenge datasets combined except for Webscope, whose access is limited under a license. All the code used for our experiments is publicly available[5].

6.1 Results for RDFS reasoning

We evaluate our system in terms of time required to calculate the full closure. We report the average and the relative deviation σ (the standard deviation divided by the average) of three runs. The results, along with the number of output

[4] `http://www.cs.vu.nl/~pmika/swc/btc.html`
[5] `https://code.launchpad.net/~jrbn/+junk/reasoning-hadoop`

triples, are presented in Table 3. Figure 6 shows the time needed for each reasoning phase. Our RDFS implementation shows very high performance: for the combined dataset of 865M triples, it produced 30B triples in less than one hour. This amounts to a total throughput of 8.77 million triples/sec. for the output and 252.000 triples/sec. for the input. These results do not include dictionary encoding, which took, as mentioned, one hour for all datasets combined. Including this time, the throughput becomes 4.27 million triples/sec. and 123.000 triples/sec. respectively, which to the best of our knowledge, still outperforms any results reported both in the literature [11] and on the Web[6].

Besides absolute performance, an important metric in parallel algorithms is how performance scales with additional compute nodes. Table 4 shows the speedup gained with increasing number of nodes and the resulting efficiency, on the Falcon and DBpedia datasets. Similar results hold for the other datasets. To the best of our knowledge, the only published speedup results for distributed reasoning on a dataset of this size can be found in [14]; for both datasets, and all numbers of nodes, our implementation outperforms this approach.

The speedup results are also shown in Figure 5. They show that our high throughput rates are already obtained when utilising only 16 compute nodes. We attribute the decreasing efficiency on larger numbers of nodes to the fixed Hadoop overhead for starting jobs on nodes: on 64 nodes, our computation per node is not big enough to compensate platform overhead.

Figure 6 shows the division of runtime over the computation phase from Figure 4, and confirms the widely-held intuition that subclass-reasoning is the most expensive part of RDFS inference on real-world datasets.

We have verified the correctness of our implementation on the (small) Wordnet dataset. We have not stored the output of our algorithm: 30B triples (each of them occupying 25 bytes using our dictionary encoding) produce 750GB of data. Mapping these triples back to the original terms would require approx. 500 bytes per triple, amounting to some 15TB of disk space.

In a distributed setting load balancing is an important issue. The Hadoop framework dynamically schedules tasks to optimize the node workload. Furthermore, our algorithms are designed to prevent load balancing problems by intelligently grouping triples (see sections 5.1 and 5.2). During experimentation, we did not encounter any load balancing issues.

6.2 Results for OWL reasoning

We have also encoded the OWL Horst rules [9] to investigate whether our approach can be extended for efficient OWL reasoning. The OWL Horst rules are more complex than the RDFS rules, and we need to launch more jobs to compute the full closure. Due to space restrictions, we refer to [16], for the algorithms and the implementation.

On the LUBM(50) benchmark dataset [5], containing 7M triples, we compute the OWL Horst closure on 32 nodes in about 3 hours, resulting in about

[6] e.g. at `esw.w3.org/topic/LargeTripleStores`

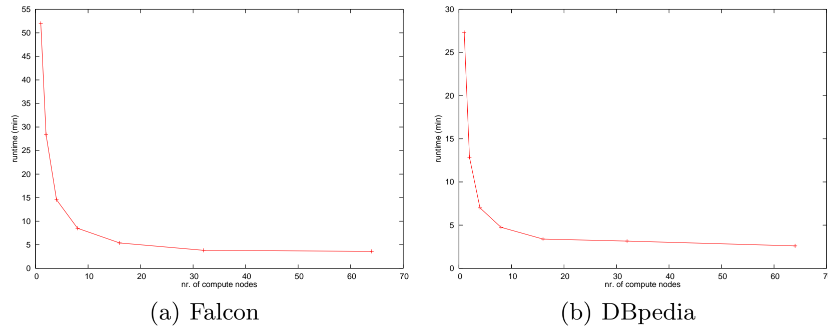
(a) Falcon

nodes	runtime (s)	speedup	efficiency
1	3120	1	1
2	1704	1.83	0.92
4	873	3.57	0.89
8	510	6.12	0.76
16	323	9.65	0.60
32	229	13.61	0.43
64	216	14.45	0.23

(b) DBpedia

nodes	runtime (s)	speedup	efficiency
1	1639	1	1
2	772	2.12	1.06
4	420	3.9	0.98
8	285	5.76	0.72
16	203	8.07	0.5
32	189	8.69	0.27
64	156	10.53	0.16

Table 4. Speedup with increasing number of nodes



(a) Falcon (b) DBpedia

Fig. 5. Speedup with increasing number of nodes

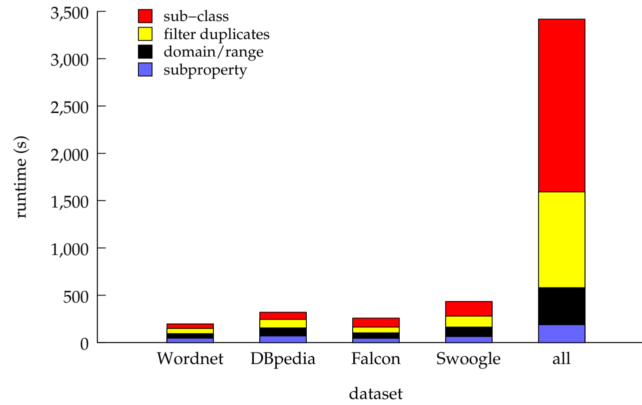


Fig. 6. Detailed reasoning time (per phase) for various datasets (32 nodes)

13M triples. In comparison, the RDFS closure on the same dataset is computed in about 10 minutes, resulting in about 8.6M triples. On a real-world dataset (Falcon, 35M triples) we stopped OWL Horst inference after 12 hours, at which point more than 130 MapReduce jobs had been launched, and some 3.8B triples had been derived. Clearly, our implementation on OWL Horst has room for optimisation; on RDFS, our optimisations drastically reduced computation time.

6.3 Discussion

Some datasets produce very large amounts of output. For example, the closure of the Swoogle and Falcon datasets is around 20× the original data. We attribute these differences to the content and quality of these datasets: data on the Web contains cycles, override definitions of standard vocabularies, etc. Instead of applying the standard RDFS and OWL rules, Hogan *et al.* [7] propose to only consider "authoritative" statements to prevent this data explosion during reasoning. In this paper, we did not focus on data quality. To avoid the observed inference explosion, the approach from [7] can be added to our algorithms.

As explained, the presented algorithm performs incomplete RDFS reasoning. We ignore RDF axiomatic triples because this is widely accepted practice and in line with most of the existing reasoners. We omit the rules with one antecedent since parallelizing their application is trivial and they are commonly ignored by reasoners as being uninteresting. If standard compliance is sought, these rules can be implemented with a single *map* over the final data, which very easy to parallelise and should not take more than some minutes. Similarly, we have ignored the rule concerning container-membership properties since these occur very rarely: in all 865M triples, there are only 10 container-membership properties, of which one is in the `example.org` namespace and two override standard RDFS. If needed, membership properties can be implemented in the same way as the subproperty-phase (albeit on much less data), which takes approximately 3 minutes to execute on the complete dataset, as seen in Figure 6.

7 Conclusion

MapReduce is a widely used programming model for data processing on large clusters, and it is used in different contexts to process large collections of data. Our purpose was to exploit the advantages of this programming model for Semantic Web reasoning; a non-trivial task given the high data correlation. We have shown a scalable implementation of RDFS reasoning based on MapReduce which can infer 30 billion triples from a real-world dataset in less than two hours, yielding an input and output throughput of 123.000 triples/second and 4.27 million triples/second respectively. To the best of our knowledge, our system outperforms any other published approach. To achieve this, we have presented some non-trivial optimisations for encoding the RDFS ruleset in MapReduce. We have evaluated the scalability of our implementation on a cluster of 64 compute nodes using several real-world datasets.

A remaining challenge is to apply the same techniques successfully to OWL-Horst reasoning. Our first experiments have shown this to be more challenging.

We would like to thank Christian Rossow for reviewing our work. This work was supported by the LarKC project (EU FP7-215535).

References

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

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

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pp. 137–147. 2004.

[4] Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable distributed ontology reasoning using DHT-based partitioning. In *ASWC*, 2008.

[5] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3:158–182, 2005.

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

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

[8] A. Hogan, A. Polleres, and A. Harth. Saor: Authoritative reasoning for the web. In *ASWC*, 2008.

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

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

[11] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM – a pragmatic semantic repository for OWL. In *Web Information Systems Engineering (WISE) Workshops*, pp. 182–192, 2005.

[12] P. Mika and G. Tummarello. Web semantics in the clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[13] B. MacCartney, S. A. McIlraith, E. Amir, and T. Uribe. Practical partition-based theorem proving for large knowledge bases. In *IJCAI*, 2003.

[14] E. Oren, S. Kotoulas, *et al.* Marvin: A platform for large-scale analysis of Semantic Web data. In *Int. Web Science conference*, 2009.

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

[16] J. Urbani. *Scalable Distributed RDFS/OWL Reasoning using MapReduce.* Master's thesis, Vrije Universiteit Amsterdam, 2009. Available from `http://www.few.vu.nl/~jui200/thesis.pdf`.

[17] J. Zhou, L. Ma, Q. Liu, L. Zhang, *et al.* Minerva: A scalable OWL ontology storage and inference system. In *ASWC*, pp. 429–443. 2006.