Large Scale Fuzzy pD^* Reasoning using MapReduce

Chang Liu[1] Guilin Qi[2] Haofen Wang[1] Yong Yu[1]

[1]Shanghai Jiaotong University, China
{liuchang,whfcarter,yyu}@apex.sjtu.edu.cn
[2]Southeast University, China
gqi@seu.edu.cn

Abstract. The MapReduce framework has proved to be very efficient for data-intensive tasks. Earlier work has tried to use MapReduce for large scale reasoning for pD^* semantics and has shown promising results. In this paper, we move a step forward to consider scalable reasoning on top of semantic data under fuzzy pD^* semantics (i.e., an extension of OWL pD^* semantics with fuzzy vagueness). To the best of our knowledge, this is the first work to investigate how MapReduce can help to solve the scalability issue of fuzzy OWL reasoning. While most of the optimizations used by the existing MapReduce framework for pD^* semantics are also applicable for fuzzy pD^* semantics, unique challenges arise when we handle the fuzzy information. We identify these key challenges, and propose a solution for tackling each of them. Furthermore, we implement a prototype system for the evaluation purpose. The experimental results show that the running time of our system is comparable with that of WebPIE, the state-of-the-art inference engine for scalable reasoning in pD^* semantics.

1 Introduction

The Resource Description Framework (RDF) is one of the major representation standards for the Semantic Web. RDF Schema (RDFS) is used to describe vocabularies used in RDF descriptions. However, RDF and RDFS only provide a very limited expressiveness. In [3], a subset of Ontology Web Language (OWL) vocabulary (e.g., owl:sameAs) was introduced, which extends the RDFS semantics to the pD^* fragment of OWL. Unlike the standard OWL (DL or Full) semantics which provides the full if and only if semantics, the OWL pD^* fragment follows RDF(S)'s if semantics. That is, the OWL pD^* fragment provides a complete set of entailment rules, which guarantees that the entailment relationship can be determined within polynomial time under a non-trivial condition (if the target graph is ground). It has become a very promising ontology language for the Semantic Web as it trades off the high computational complexity of OWL Full and the limited expressiveness of RDFS.

Recently, there is an increasing interest in extending RDF to represent vague information on Web. Fuzzy RDF allows us to state to a certain degree, a triple is true. For example, $(Tom, eat, pizza)$ is true with degree at least 0.8. However, Fuzzy RDF (or fuzzy RDFS) has limited expressive power to represent information in some real life applications of ontologies, such as biomedicine and multimedia. In [4], we extended the OWL pD^* fragment with fuzzy semantics to provide more expressive power than fuzzy RDF(S). In that work, we focused on some theoretical problems, such as the complexity issues, without providing an efficient reasoning algorithm for the new semantics. Since fuzzy pD^* semantics is targeted to handle large scale semantic data, it is critical to provide a scalable reasoning algorithm for it.

Earlier works (e.g. [12]) have proved that MapReduce is a very efficient framework to handle the computation of the closure containing up to 100 billion triples under pD^* semantics. One may wonder if it is helpful to scalable reasoning in fuzzy pD^* semantics. It turns out that this is a non-trivial problem as the computation of the closure under fuzzy pD^* semantics requires the computation of the *Best Degree Bound (BDB)* of each triple. The BDB of a triple is the greatest lower bound of the fuzzy degrees of this triple. Although most of the optimizations for reasoning with MapReduce in pD^* semantics are also applicable for the fuzzy pD^* semantics, unique challenges arise when we handle the fuzzy information.

In this paper, we first identify some challenges to apply the MapReduce framework to deal with reasoning in fuzzy pD^* semantics. We then propose an algorithm for fuzzy $pD*$ reasoning by separately considering fuzzy D rules and fuzzy p rules. After that, we propose the *map* function and the *reduce* function for several fuzzy pD^* rules that may cause difficulties. Finally, we implement a prototype system to evaluate these optimizations. The experimental results show that the running time of our system is comparable with that of WebPIE [12] which is the state-of-the-art inference engine for OWL pD^* fragment.

2 Background Knowledge

In this section, we first introduce the fuzzy pD^* entailment rule set in Section 2.1, then explain the MapReduce framework for reasoning in OWL pD^* fragment in Section 2.2.

2.1 Fuzzy RDF and Fuzzy pD^* Reasoning

A fuzzy RDF graph is a set of fuzzy triples which are in form of $t[n]$. Here t is a triple, and $n \in (0, 1]$ is the fuzzy degree of t.

Fuzzy pD^* semantics, given in [4], extends pD^* semantics with fuzzy semantics so that there is a complete and sound entailment rule set in fuzzy OWL pD^* fragment. We list part of them in Table 1 by excluding some naive rules. The notion of a (partial) closure can be easily extended to the fuzzy case.

The key notion in fuzzy pD^* semantics is called the *Best Degree Bound (BDB)* of a triple. The BDB n of an arbitrary triple t from a fuzzy RDF graph G is defined to be the largest fuzzy degree n such that $t[n]$ can be derived from

Table 1. Difficult part of fuzzy pD^*-entailment rules

Condition	Conclusion	
f-rdfs2	$(p, \texttt{domain}, u)[n]$ $(v, p, w)[m]$	$(v, \texttt{type}, u)[n \otimes m]$
f-rdfs3	$(p, \texttt{range}, w)[n]$ $(v, p, w)[m]$	$(v, \texttt{type}, w)[n \otimes m]$
f-rdfs5	$(v, \texttt{subPropertyOf}, w)[n]$ $(w, \texttt{subPropertyOf}, u)[m]$	$(v, \texttt{subPropertyOf}, u)[n \otimes m]$
f-rdfs7x	$(p, \texttt{subPropertyOf}, q)[n]$ $(v, p, w)[m]$	$(v, q, w)[n \otimes m]$
f-rdfs9	$(v, \texttt{subClassOf}, w)[n]$ $(u, \texttt{type}, v)[m]$	$(u, \texttt{type}, w)[n \otimes m]$
f-rdfs11	$(v, \texttt{subClassOf}, w)[n]$ $(w, \texttt{subClassOf}, u)[m]$	$(v, \texttt{subClassOf}, u)[n \otimes m]$
f-rdfs12	$(v, \texttt{type}, \texttt{ContainerMembershipProperty})[n]$	$(v, \texttt{subPropertyOf}, \texttt{member})[n]$
f-rdfs13	$(v, \texttt{type}, \texttt{Datatype})[n]$	$(v, \texttt{subClassOf}, \texttt{Literal})[1]$
f-rdfp1	$(p, \texttt{type}, \texttt{FunctionalProperty})[n]$ $(u, p, v)[m]$ $(u, p, w)[l]$	$(v, \texttt{sameAs}, w)[l \otimes m \otimes n]$
f-rdfp2	$(p, \texttt{type}, \texttt{InverseFunctionalProperty})[n]$	
	$(u, p, w)[m]$ $(v, p, w)[l]$	$(u, \texttt{sameA}, v)[l \otimes m \otimes n]$
f-rdfp3	$(p, \texttt{type}, \texttt{SymmetricProperty})[n]$ $(v, p, w)[m]$	$(w, p, v)[n \otimes m]$
f-rdfp4	$(p, \texttt{type}, \texttt{TransitiveProperty})[n]$ $(u, p, v)[m]$ $(v, p, w)[l]$	$(u, p, w)[n \otimes m \otimes l]$
f-rdfp5(ab)	$(v, p, w)[n]$	$(v, \texttt{sameAs}, v)[1]$, $(w, \texttt{sameAs}, w)[1]$
f-rdfp6	$(v, \texttt{sameAs}, w)[n]$	$(w, \texttt{sameAs}, v)[n]$
f-rdfp7	$(u, \texttt{sameAs}, v)[n]$ $(v, \texttt{sameAs}, w)[m]$	$(u, \texttt{sameAs}, w)[n \otimes m]$
f-rdfp8ax	$(p, \texttt{inverseOf}, q)[n]$ $(v, p, w)[m]$	$(w, q, v)[n \otimes m]$
f-rdfp8bx	$(p, \texttt{inverseOf}, q)[n]$ $(v, q, w)[m]$	$(w, p, v)[n \otimes m]$
f-rdfp9	$(v, \texttt{type}, \texttt{Class})[n]$ $(v, \texttt{sameAs}, w)[m]$	$(v, \texttt{subClassOf}, w)[m]$
f-rdfp10	$(p, \texttt{type}, \texttt{Property})[1]$ $(p, \texttt{sameAs}, q)[m]$	$(p, \texttt{subPropertyOf}, q)[m]$
f-rdfp11	$(u, p, v)[n]$ $(u, \texttt{sameAs}, u')[m]$ $(v, \texttt{sameAs}, v')[l]$	$(u', p, v')[n \otimes m \otimes l]$
f-rdfp12(ab)	$(v, \texttt{equivalentClass}, w)[n] \Rightarrow (v, \texttt{subClassOf}, w)[n]$,$(w, \texttt{subClassOf}, w)[n]$	
f-rdfp12c	$(v, \texttt{subClassOf}, w)[n]$ $(w, \texttt{subClassOf}, v)[m]$	$(v, \texttt{equivalentClass}, w)[\min(n, m)]$
f-rdfp13(ab)	$(v, \texttt{equivalentProperty}, w)[n] \Rightarrow (v, \texttt{subPropertyOf}, w)[n]$, $(w, \texttt{subPropertyOf}, w)[n]$	
f-rdfp13c	$(v, \texttt{subPropertyOf}, w)[n]$ $(w, \texttt{subPropertyOf}, v)[m]$	$(v, \texttt{equivalentClass}, w)[\min(n, m)]$
f-rdfp14a	$(v, \texttt{hasValueOf}, w)[n]$ $(v, \texttt{onProperty}, p)[m]$ $(u, p, w)[l]$	$(u, \texttt{type}, v)[n \otimes m \otimes l]$
f-rdfp14bx	$(v, \texttt{hasValueOf}, w)[n]$ $(v, \texttt{onProperty}, p)[m]$ $(u, \texttt{type}, v)[l]$	$(u, p, w)[n \otimes m \otimes l]$
f-rdfp15	$(v, \texttt{someValueFrom}, w)[n]$ $(v, \texttt{onProperty}, p)[m]$	
	$(u, p, x)[l]$ $(x, \texttt{type}, w)[k]$	$(u, \texttt{type}, v)[n \otimes m \otimes l \otimes k]$
f-rdfp16	$(v, \texttt{allValuesfrom}, w)[m]$ $(v, \texttt{onProperty}, p)[n]$	
	$(u, \texttt{type}, v)[l]$ $(u, p, x)[k]$	$(x, \texttt{type}, w)[n \otimes m \otimes l \otimes k]$

G by applying fuzzy pD^*-entailment rules, or 0 if no such fuzzy triple can be derived.

2.2 MapReduce Algorithm for pD^* Reasoning

MapReduce is a programming model introduced by Google for large scale data processing [1]. A MapReduce program is composed of two user-specified functions, *map* and *reduce*. When the input data is appointed, the *map* function scans the input data and generates intermediate key/value pairs. Then all pairs of key and value are partitioned according to the key and each partition is processed by a *reduce* function.

We use an example to illustrate how to use a MapReduce program to apply a rule. Here, we consider Rule rdfs2 which reads:

$$(p, \texttt{domain}, u), (v, p, w) \Rightarrow (v, \texttt{type}, u)$$

The *map* function scans the data set, and checks every triple if it has the form $(p, \texttt{domain}, u)$ or (v, p, w). If a triple has the form $(p, \texttt{domain}, u)$, then the *map* function generates an output (key=p, value={flag='L', u}). While a triple in the form of (v, p, w) is scanned, the *map* function generates an output (key=p, value={flag='R', v}). The *reduce* function gets all outputs of the *map* function that share the same key together. Then it enumerates all values with flag 'L' to get all u and enumerates all values with flag 'R' to get all v. For each pair of u and v, the *reduce* function generates a new triple $(v, \texttt{type}, u)$ as output.

There are several key factors to make a MapReduce program efficient. Firstly, since the *map* function operates on single pieces of data without dependencies, partitions can be created arbitrarily and can be scheduled in parallel across many nodes. Secondly, the *reduce* function operates on an iterator of values since the set of values is typically too large to fit in memory. So the reducer should treat the values as a stream instead of a set. Finally, all the outputs of mappers sharing the same key will be processed by the same *reduce* function. Therefore, the reducer that processes one popular key will run very slowly. The mappers' output keys should be carefully designed to ensure that the sizes of all partitions should be balanced.

Due to these reasons, the naive implementations of rules may be very inefficient. In [12] and [13], several optimizations are proposed that improve the performance of inference in OWL pD^* fragment significantly. We list them as follows.

- **Loading schema triples in memory.** Since the set of schema triples is generally small enough to fit in memory, when performing a join over schema triples, we can load them into memory. Then, the join can be performed directly between the loaded data and the in-memory schema triples.
- **Data grouping to avoid duplicates.** Some RDFS rules may generate duplicates. However, using carefully designed algorithms, such duplicates can be avoided.
- **Ordering the RDFS rule applications.** Arbitrarily applying the rules will result in a fixpoint iteration. For RDFS rules, such a fixpoint iteration can be avoided by applying rules in a specific order.
- **Transitive algorithm.** An efficient algorithm to calculate the transitive closure is designed, which will produce a minimal amount of duplicates and minimize the number of iterations.
- **Sameas algorithm.** For OWL pD^* fragment, [12] uses the canonical representation to deal with the **sameas** rules. This method greatly reduces both the computation time and the space required.
- **someValuesFrom and allValuesFrom algorithm.** In both rules involve someValuesFrom and allValuesFrom, three joins among four triples are needed. However, two of the four triples are schema triples so that they can be loaded into memory. Furthermore by choosing the output key, the *map* function will generate balanced partitions for the *reduce* function.

3 MapReduce Algorithm for Fuzzy pD^* Reasoning

In this section, we first illustrate how to use a MapReduce program to apply a fuzzy rule. Then, we give an overview of the challenges in fuzzy pD^* reasoning when applying the MapReduce framework. Finally, we present our solutions to handle these challenges.

Algorithm 1 map function for rule f-rdfs2

```
Input: key, triple
if triple.predicate == 'domain' then
    emit({p=triple.subject}, {flag='L', u=triple.object, n=triple.degree};
end if
emit({p=triple.predicate}, {flag='R', v=triple.subject, m=triple.degree};
```

Algorithm 2 reduce function for rule f-rdfs2

```
Input: key, iterator values
unSet.clear();
vmSet.clear();
for value $\in$ values do
    if value.flag == 'L' then
        unSet.update(value.u, value.n);
    else
        vmSet.update(value.v, value.m);
    end if
end for
for $i \in$ unSet do
    for $j \in$ vmSet do
        emit(null, triple(i.u, 'type', j.v, i.n$\otimes$j.m));
    end for
end for
```

3.1 Naive MapReduce algorithm for fuzzy rules

We consider rule f-rdfs2 to illustrate our naive MapReduce algorithms:

$$(p, \texttt{domain}, u)[n], (v, p, w)[m] \Rightarrow (v, \texttt{type}, u)[n \otimes m]$$

In this rule, we should find all fuzzy triples that are either in the form of $(p, \texttt{domain}, u)[n]$ or in the form of $(v, p, w)[m]$. A join should be performed over the variable p. The *map* and *reduce* functions are given in Algorithms 1 and 2 respectively. In the *map* function, when a fuzzy triple is in the form of $(p, \texttt{domain}, u)[n]$ (or $(v, p, w)[m]$), the mapper emits p as the key and u (or v) along with the degree n (or m) as the value. The reducer can use the flag in the mapper's output value to identify the content of the value. If the flag is 'L' (or 'R'), the content of the value is the pair (u, n) (or the pair (v, m)). The reducer uses two sets to collect all the u, n pairs and the v, m pairs. After all pairs are collected, the reducer enumerates pairs (u, n) and (v, m) to generate $(u, \texttt{type}, v)[n \otimes m]$ as output.

3.2 Challenges in Fuzzy pD^* Reasoning

Even though the fuzzy pD^* entailment rules are quite similar to the pD^* rules, several difficulties arise when we calculate the BDB for each triple by applying the MapReduce framework. We summarize these challenges as follows:

Ordering the rule applications. In fuzzy pD^* semantics, the reasoner might produce the duplicated triple with different fuzzy degrees before the fuzzy B-DB triple is derived. For example, suppose the data set contains a fuzzy triple $t[m]$, when we derive a fuzzy triple $t[n]$ with $n > m$, a duplicate is generated. When duplicates are generated, we should employ a duplicate deleting program to reproduce the data set to ensure that only the fuzzy triples with maximal degrees are in the data set. Different rule applications' order will result in different number of such duplicates. To achieve the best performance, we should choose a proper order to reduce the number of duplicates. For instance, the subproperty rule (f-rdfs7x) should be applied before the domain rule (f-rdfs2); and the equivalent class rule (f-rdfp12(abc)) should be considered together with the subclass rule (f-rdfs9, f-rdfs10). The solution for this problem will be discussed in Section 3.3.

Shortest path calculation. In OWL pD^* fragment, the three rules, rdfs5 (subproperty), rdfs11 (subclass) and rdfp4 (transitive property) are essentially used to calculate the transitive closure over a subgraph of the RDF graph. In fuzzy OWL pD^*, when we treat each fuzzy triple as a weighted edge in the RDF graph, then calculating the closure by applying these three rules is essentially a variation of the all-pairs shortest path calculation problem. We have to find out efficient algorithms for this problem. In Section 3.4, we will discuss the solutions for rules f-rdfs5 and f-rdfs11, while discuss rule f-rdfp4 in Section 3.5.

Sameas rule. For OWL pD^* fragment, the traditional technique to handle the semantics of `sameas` is called canonical representation. Rules rdfp6 and rdfp7 enforce that `sameas` is a symmetric and transitive property, thus the `sameas` closure obtained by applying these two rules is composed of several complete subgraphs. The instances in the same subgraph are all synonyms, so we can assign a unique key, which we call the canonical representation, to all of them. Replacing all the instances with its unique key results in a more compact representation of the RDF graph without loss of completeness for inference.

However, in the fuzzy pD^* semantics, we cannot choose such a canonical representation as illustrated by the following example. Suppose we use the *min* as the t-norm function. Given a fuzzy RDF graph G containing seven triples:

$(a, \texttt{sameas}, b)[0.8]$ $(b, \texttt{sameas}, c)[0.1]$ $(c, \texttt{sameas}, d)[0.8]$
$(a, \texttt{range}, r)[0.9]$ $(u, b, v)[0.9]$ $(c, \texttt{domain}e)[1]$ $(u', d, v')[0.9]$

From this graph, we can derive $(v, \texttt{type}, r)[0.8]$. Indeed, we can derive $(b,$ $\texttt{range}, r)[0.8]$ by applying rule f-rdfp11 over $(a, \texttt{sameas}, b)[0.8]$, $(a, \texttt{range}, r)[0.9]$ and $(r, \texttt{sameas}, r)[1.0]$. Then we can apply rule f-rdfs3 over $(b, \texttt{range}, r)[0.8]$ and $(u, b, v)[0.9]$ to derive $(v, \texttt{type}, r)[0.8]$.

In this graph, four instances, a, b, c and d are considered as synonyms in the classical pD^* semantics. Suppose we choose c as the canonical representation, then the fuzzy RDF graph is converted into the following graph G' containing four fuzzy triples:

$(c, \texttt{range}, r)[0.1]$ $(u, c, v)[0.1]$ $(c, \texttt{domain}e)[1]$ $(u', c, v')[0.8]$

From this graph, we can derive the fuzzy triple $(v, \texttt{type}, r)[0.1]$, and this is a fuzzy BDB triple from G', which means we cannot derive the fuzzy triple, $(v, \texttt{type}, r)[0.8]$. The reason is that after replacing a and b with c, the fuzzy information between a and b, e.g. the fuzzy triple $(a, \texttt{sameas}, b)[0.8]$, is missing. Furthermore, no matter how we choose the canonical representation, some information will inevitably get lost during the replacement. We will discuss the solution for this problem in Section 3.6.

3.3 Overview of the reasoning algorithm

Our main reasoning algorithm is Algorithm 3, which can be separated into two phases: the first phase (line 3) applies the fuzzy D rules (from f-rdfs1 to f-rdfs13), and the second phase (lines 7 to line 9) applies the fuzzy p rules (from rdfp1 to rdfp16). However, since some fuzzy p rules may generate some fuzzy triples having effect on fuzzy D rules, we execute these two phases iteratively (line 2 to line 11) until a fix point is reached (line 4 to line 6).

In the first phase, we consider the following order of rule applications such that we can avoid a fix point iteration. Firstly, we apply the property inheritance rules (f-rdfs5 and f-rdfs7), so that domain rule (f-rdfs2) and range rule (f-rdfs3) can be applied consecutively without loosing any important fuzzy triples. Then the class inheritance rules (f-rdfs9 and f-rdfs11) along with the rest rules are applied together. Similar techniques have been used in [13] for RDFS. Compared with [13], our algorithm relies on the computation of rules f-rdfs5, f-rdfs7, f-rdfs9 and f-rdfs11. We will discuss this point in the next section.

For the fuzzy p rules, there is no way to avoid a fixpoint iteration. So we employ an iterative algorithm to calculate the p closure. In each iteration, the program can be separated into five steps. In the first step, all non-iterative rules (rules f-rdfp1, 2, 3, 8) are applied. The second step processes the transitive property (rule f-rdfp4) while the `sameas` rules (rule f-rdfp6, 7, 10, 11) are applied in the third step. The rules related to `equivalentClass`, `equivalentProperty` and `hasValue` are treated in the fourth step, because we can use the optimizations for reasoning in OWL pD^* fragment to compute the closure of these rules in a non-iterative manner. The `someValuesFrom` and `allValuesFrom` rules are applied in the fifth step which needs a fixpoint iteration. The first step and the last two steps can employ the same optimization discussed in [12]. We will discuss the solution to deal with transitive property in Section 3.5. Finally the solution to tackle `sameas` rules will be discussed in Section 3.6.

3.4 Calculating `subClassOf` and `subPropertyOf` Closure

Rules f-rdfs5, 6, 7, f-rdfp13(abc) process the semantics of `subPropertyOf` property while rules f-rdfs9, 10, 11 and f-rdfp12(abc) mainly concern the semantics of `subClassOf` property. Since they can be disposed similarly, we only discuss the rules that are relevant to `subClassOf`. Since f-rdfs10 only derives a triple $(v, \texttt{subClassOf}, v)[1]$ which will have no affect on other rules, we only consider rules f-rdfs5, 7 and f-rdfp12(abc).

Algorithm 3 Fuzzy pD^* reasoning

```
1: first_time = true;
2: while true do
3:    derived = apply_fD_rules();
4:    if derived == 0 and not first_time then
5:       break;
6:    end if;
7:    repeat
8:       derived = apply_fp_rules();
9:    until derived == 0;
10:   first_time = false;
11: end while
```

We call the triples in the form of (u, `subClassOf`, v)[n] to be `subClassOf` triples. The key task is to calculate the closure of `subClassOf` triples by applying rule f-rdfs11. Since the set of `subClassOf` triples is relatively small, we can load them into memory. We can see that calculating the `subClassOf` closure by applying rule f-rdfs11 is indeed a variation of the all-pairs shortest path calculation problem, according to the following property:

Property 1 *For any fuzzy triple in the form of (u,* `subClassOf`*, v)[n] that can be derived from the original fuzzy RDF graph by only applying rule f-rdfs11, there must be a chain of classes $w_0 = u, w_1, ..., w_k = v$ and a list of fuzzy degrees $d_1, ..., d_k$ where for every $i = 1, 2, ..., k$, $(w_{i-1},$* `subClassOf`*$, w_i)[d_k]$ is in the original fuzzy graph and $n = d_1 \otimes d_2 \otimes ... \otimes d_k$.*

So we can use the FloydCWarshall style algorithm given in Algorithm 4 to calculate the closure. In the algorithm, I is the set of all the classes, and $w(i, j)$ is the fuzzy degree of triple $(i,$ `subClassOf`$, j)$. The algorithm iteratively update the matrix w. When it stops, the subgraph represented by the matrix $w(i, j)$ is indeed the `subClassOf` closure.

The worst-case running complexity of the algorithm is $O(|I|^3)$, and the algorithm uses $O(|I|^2)$ space to store the matrix w. When $|I|$ goes large, this is unacceptable. However, we can use nested hash map instead of 2-dimension arrays to only store the positive matrix items. Furthermore, since $0 \otimes n = n \otimes 0 = 0$, in line 2 and line 3, we only enumerate those i and j where $w(k, i) > 0$ and $w(k, j) > 0$. In this case, the running time of the algorithm will be greatly reduced.

After the `subClassOf` closure is computed, rules f-rdfs9 and f-rdfs11 can be applied only once to derive all fuzzy triples: for rule f-rdfs9 (or f-rdfs11), when we find a fuzzy triple $(i,$ `type`$, v)[n]$ (or $(i,$ `subClassOf`$, v)[n]$), we enumerate all classes j so that $w(v, j) > 0$ and output a fuzzy triple $(i,$ `type`$, j)[n \otimes w(v, j)]$ (or $(i,$ `subClassOf`$, j)[n \otimes w(v, j)]$).

For rule f-rdfp12(abc), since `equivalentClass` triples are also schema triples, we load them into memory and combine them into the `subClassOf` graph. Specifically, when we load a triple $(i,$ `equivalentClass`$, j)[n]$ into memory, if

Algorithm 4 Calculate the `subClassOf` closure

```
1: for $k \in I$ do
2:     for $i \in I$ and $w(i, k) > 0$ do
3:         for $j \in I$ and $w(k, j) > 0$ do
4:             if $w(i, k) \otimes w(k, j) > w(i, j)$ then
5:                 $w(i, j) = w(i, k) \otimes w(k, j)$;
6:             end if
7:         end for
8:     end for
9: end for
```

Algorithm 5 map function for rule f-rdfp4

```
Input: length, triple=(subject, predicate, object)[degree], n
  if getTransitiveDegree(predicate) == 0 then
      return;
  end if
  if length ==$2^{n-2}$ or length == $2^{n-1}$ then
      emit({predicate, object}, {flag=L, length, subject, degree};
  end if
  if length > $2^{n-2}$ and length ≤ $2^{n-1}$ then
      emit({predicate, subject}, {flag=R, length, object, degree};
  end if
```

$n > w(i,j)$ (or $n > w(j,i)$), we update $w(i,j)$ (or $w(j,i)$) to be n . After the closure is calculated, two fuzzy triples $(i, \texttt{equivalentClass}, j)[n]$ and $(j, \texttt{equivalentClass}, i)[n]$ are output for each pair of classes $i, j \in I$, if $n = \min(w(i,j), w(j,i)) > 0$.

3.5 Transitive Closure for `TransitiveProperty`

The computation of the transitive closure by applying rule f-rdfp4 is essentially calculating the all-pairs shortest path on the instance graph. To see this point, we consider the following property:

Property 2 *Suppose there is a fuzzy triple* $(p, \texttt{Type}, \texttt{TransitiveProperty})[n]$ *in the fuzzy RDF graph G, and* $(a, p, b)[m]$ *is a fuzzy triple that can be derived from G using only rule f-rdfp4. Then there must be a chain of instances* $u_0 = a, u_1, ..., u_k = b$ *and a list of fuzzy degree* $d_1, ..., d_k$ *such that* $m = d_1 \otimes n \otimes d_2 \otimes ... \otimes n \otimes d_k$*, and for every* $i = 1, 2, ..., k$, $(u_{i-1}, p, u_p)[d_i]$ *is in the original fuzzy RDF graph. Furthermore, in one of such chains,* $u_i \neq u_j$*, if* $i \neq j$ *and* $i, j \geq 1$.

We use an iterative algorithm to calculate this transitive closure. In each iteration, we execute a MapReduce program using Algorithm 5 as the *map* function and Algorithm 6 as the *reduce* function.

We use `getTransitiveDegree(p)` function to get the maximal n such that $(p, \texttt{type}, \texttt{TransitiveProperty})[n]$ is in the graph. Since these triples are schema

Algorithm 6 reduce function for rule f-rdfp4

```
Input: key, iterator values
  left.clear();
  right.clear();
  for value ∈ values do
    if value.flag == 'L' then
      left.update(value.subject, {value.degree, value.length});
    else
      right.update(value.object, {value.degree, value.length});
    end if
  end for
  for $i$ ∈ left do
    for $j$ ∈ right do
      newLength = $i$.length + $j$.length;
      emit(newLength, triple($i$.subject, key.predicate, $j$.object,
        $i.degree \otimes j.degree \otimes$ getTransitiveDegree($key.predicate$)));
    end for
  end for
```

triples, we can load them into memory before the mappers and reducers are executed. In the *map* function, n is the number of iterations that the transitive closure calculation algorithm already executes. Since for any fuzzy triple $(a, p, b)[m]$, there is at least one chain $u_0 = a, u_1, ..., u_k = b$ according to Property 2, we use variable *length* to indicate the length of the shortest chain of $(a, p, b)[m]$. At the beginning of the algorithm, for every triple $(a, p, b)[n]$ in the fuzzy RDF graph, *length* is assigned to be one.

If $(a, p, b)[m]$ has a chain $u_0, u_1, ..., u_k$ with length k, and it can be derived from $(a, p, t)[m_1]$ and $(t, p, b)[m_2]$ in the n-th iteration, then we have $m_1 + m_2 = m$, $m_1 = 2^{n-2}$ or 2^{n-1}, and $2^{n-2} < m_2 \leq 2^{n-1}$. We can prove the integer equation $m_1+m_2 = m$ has a unique solution satisfying $m_1 = 2^{n-2}$ or $m_1 = 2^{n-1}$, and $2^{n-2} < m_2 \leq 2^{n-1}$. Thus for such a chain, the triple $(a, p, b)[m]$ will be generated only once. As a consequence, a fuzzy triple will be generated at most as many times as the number of chains it has. In most circumstances, every fuzzy triple will be generated only once.

Furthermore, based on the above discussion, if a fuzzy triple $(a, p, b)[m]$ has a chain with length $2^{n-1} < l \leq 2^n$, it will be derived within n iterations. As a consequence, the algorithm will terminate within $\log N$ iterations where N is the number of all instances in the graph.

3.6 Handling `SameAs` Closure

The rules related to `sameas` are f-rdfp5(ab), 6, 7, 9, 10 and 11. Rules f-rdfp5(ab) are naive rules which can be implemented directly. The conclusion of Rule f-rdfp9 can be derived by applying rules f-rdfs10 and f-rdfp11. Rule f-rdfp10 allows replacing the predicate with its synonyms. Thus we only consider the rules f-rdfp6 and f-rdfp7, and the following variation of rule f-rdfp11, called f-rdfp11x:

$$(u, p, v)[n], (u, \texttt{sameas}, u')[m], (v, \texttt{sameas}, v')[l], (p, \texttt{sameas}, p')[k]$$
$$\Rightarrow (u', p', v')[n \otimes m \otimes l \otimes k]$$

The first two rules only affect the computation of sameas closure, and the rule f-rdfp11x influences the other rules' computation.

For convenience, we call a fuzzy triple in the form of $(i, \texttt{sameas}, j)[n]$ a sameas triple. We further call the sameas triples with fuzzy degree 1 the *certain* sameas *triples*, and the others with fuzzy degree less than 1 the *vague* sameas *triples*. The sameas problem is caused by those vague sameas triples. Thus for certain sameas triples, the canonical representation technique is still applicable. In real applications, such as Linking Open Data project, most of the sameas triples are certain in order to link different URIs across different datasets. Thus the traditional technique will be helpful to solve the problem.

However, the fuzzy pD^* semantics allows using sameas triples to represent the similarity information. For these triples, we must store all of them and calculate the sameas closure using rules f-rdfp6 and f-rdfp7 to ensure the inference to be complete.

Materializing the result by applying the rule f-rdfp11x will greatly expand the dataset which may cause fatal efficiency problems. To accelerate the computation, we do not apply rule f-rdfp11x directly. Instead, we modify the algorithms for other rules to consider the effect of rule f-rdfp11x.

In the following, we use rule f-rdfs2 mentioned in 3.1 as an example to illustrate the modification. In rule f-rdfs2, two fuzzy triples join on p. Considering rule f-rdfp11x, if the dataset contains a fuzzy triple $(p, \texttt{sameas}, p')[n]$, then we can make the following inference by applying f-rdfp11x and f-rdfs2:

$$(p, \texttt{domain}, u)[m], (v, p', w)[k], (p, \texttt{sameas}, p')[n] \Rightarrow (v, \texttt{type}, u)[n \otimes m \otimes k]$$

We use Algorithm 7 to replace Algorithm 1 as the *map* function. The difference is that Algorithm 7 uses a loop between line 2 and line 5 instead of line 2 in Algorithm 1. In practice, vague sameas triples are relatively few. Thus we can load them into memory and compute the sameas closure before the mappers are launched. When the mapper scans a triple in the form of $(p, \texttt{domain}, u)[m]$, the mapper looks up the sameas closure to find the set of fuzzy triples in the form of $(p, \texttt{sameas}, p')[n]$. For each pair (p', n), the mapper outputs a key p' along with a value {flag='L', u=triple.object, $m \otimes n$}. While processing key p', the reducer will receive all the values of u and $m \otimes n$. Furthermore, the reducer will receive all values of v and k outputted by the mapper in line 7. Thus the reducer will generate fuzzy triples in the form of $(v, \texttt{type}, u)[n \otimes m \otimes k]$ as desired. Similarly we can modify the algorithms for other rules to consider the effect of rule f-rdfp11x.

Finally, we discuss the sameas problem while processing the rules f-rdfp1 and f-rdfp2, since they generate sameas triples. We only discuss the rule f-rdfp1 since

Algorithm 7 map function for rules f-rdfs2 and f-rdfp11

```
Input: key, triple
1: if triple.predicate == ’domain’ then
2:     for $(subject, \texttt{sameas}, p')[n]$ is in the sameas closure do
3:         $m$ = triple.degree;
4:         emit({p=p’}, {flag=’L’, u=triple.object, $m \otimes n$};
5:     end for
6: end if
7: emit({p=triple.predicate}, {flag=’R’, v=triple.subject, k=triple.degree};
```

the other is similar. Consider a fuzzy graph G containing the following $n+1$ fuzzy triples:

$(a, p, b_1)[m_1]$ $(a, p, b_2)[m_2]$ ... $(a, p, b_n)[m_n]$

$(p, \texttt{type}, \texttt{FunctionalProperty})[k]$

By applying the rule f-rdfp1, we can derive $n(n-1)/2$ fuzzy triples in the form of $(b_i, \texttt{sameas}, b_j)[k \otimes m_i \otimes m_j]$.

4 Experiment

We implemented a prototype system based on the Hadoop framework[1], which is an open-source Java implementation of MapReduce. Hadoop uses a distributed file system, called HDFS[2] to manage executions details such as data transfer, job scheduling, and error management.

Since there is no system supporting fuzzy pD^* reasoning, we run our system over the standard LUBM data, and validate it against the WebPIE reasoner which supports inference of Horst fragment to check the correctness of our algorithms. Our system can produce the same results as WebPIE. Furthermore, we build some small fuzzy pD^* ontologies, and a naive inference system for validation purpose. Our system can produce the same results on all these ontologies.

The experiment was conducted in a Hadoop cluster containing 25 nodes. Each node is a PC machine with a 4-core, 2.66GHz, Q8400 CPU, 8GB main-memory, 3TB hard disk. In the cluster, each node is assigned three processes to run *map* tasks, and three process to run *reduce* tasks. So the cluster allows running at most 75 mappers or 75 reducers simultaneously. Each mapper and each reducer can use at most 2GB main-memory.

4.1 Datasets

Since there is no real fuzzy RDF data available, we generate synthesis fuzzy ontology, called fpdLUBM[3], for experimental purpose. Our system is based on a fuzzy extension of LUBM [2], called fLUBM, which is used for testing querying

[1] http://hadoop.apache.org/

[2] http://hadoop.apache.org/hdfs/

[3] Available at http://apex.sjtu.edu.cn/apex_wiki/fuzzypd

ability under fuzzy DL-Lite semantics in [7]. The fLUBM dataset adds two fuzzy classes, called `Busy` and `Famous`. The fuzzy degrees of how an individual belongs to these classes are generated according to the number of courses taught or taken by the individual, and the publications of the individual respectively.

However, since there is no hierarchy among these fuzzy classes, we cannot use fLUBM to test our reasoning algorithm. To tackle this problem, we further added six fuzzy classes, `VeryBusy`, `NormalBusy`, `LessBusy`, `VeryFamous`, `NormalFamous` and `LessFamous`. Given an individual i, suppose its membership degree w.r.t. class `Busy` (the fuzzy degree how i belongs to class `Busy`) is b_i. If $b_i < 0.5$, we add a fuzzy triple `(`i`, type, LessBusy)[`$b_i/0.5$`]` into the dataset; if $0.5 \leq b_i < 0.7$, we generated a fuzzy triple `(`i`, type, NormalBusy)[`$b_i/0.7$`]`; otherwise, we generate a fuzzy triple `(`i`, type, VeryBusy)[`b`]`. We added two fuzzy triples, `(LessBusy, subClassOf, Busy)[`0.5`]` and `(VeryBusy, subClassOf, Busy)[`1.0`]` to the TBox. Similarly, we can generate the fuzzy triples related to `Famous`.

We further added a transitive property call `youngerThan` to test calculation of the transitive closure. In each university ontology, we assigned a randomly generated age to each student. Then we generated n `youngerThan` triples. For each triple, we randomly chose two different students i and j satisfying $age_i < age_j$, and added a fuzzy triple `(`i`, youngerThan, `j`)[`age_i/age_j`]` into the data set.

Finally, we added a TBox triple to assert that `emailAddress` is an inverse functional property. In fact, e-mail is usually used for identifying a person online. Furthermore, for each faculty f, since we know the university from which he got his bachelor degree, we picked one email address e belonging to an undergraduate student in that university, and added a triple `(`f`, emailAddress, `e`)[`d`]` into the data set. Here we assigned the fuzzy degrees d to be either 1.0 or 0.9. Then `sameas` triples were derived using the semantics of `inverseFunctionalProperty`. We set the probability when $d = 0.9$ to be 1%, so that a small set of vague `sameas` triples can be generated. Similarly, we can generate other emailAddress triples according to the master and doctoral information similarly.

4.2 Experimental Results

Comparison with WebPIE We compared the performance of our system with that of the baseline system WebPIE[4]. We run both systems over the same dataset fpdLUBM8000. The results are shown in Table 2. Notice that the dataset is a fuzzy dataset, for WebPIE, we simply omit the fuzzy degree, and submit all crisp triples to the system. So our system (FuzzyPD) output a little more triples than WebPIE, because our system also updates the fuzzy degrees. The running time difference between our system and WebPIE is from -5 to 20 minutes. However, since a Hadoop job's execution time is affected by the statuses of the machines in the cluster, several minutes' difference between the two systems is within a

[4] We fix some bugs in the source code which will cause performance problem.

Number of Universities	Time of FuzzyPD	Time of WebPIE
1000	38.8	41.32
2000	66.97	74.57
4000	110.40	130.87
8000	215.48	210.01

Table 2. Experimental results of our system and WebPIE

Number of units	Time (minutes)	Speedup
128	38.80	4.01
64	53.15	2.93
32	91.58	1.70
16	155.47	1.00

Table 3. Scalability over number of mappers

Number of universities	Input (MTriples)	Output (MTriples)	Time (minutes)	Throughput (KTriples/second)
1000	155.51	92.01	38.8	39.52
2000	310.71	185.97	66.97	46.28
4000	621.46	380.06	110.40	57.37
8000	1243.20	792.54	215.50	61.29

Table 4. Scalability over data volume

rational range. Thus we conclude that our system is comparable with the state-of-the-art inference system.

Scalability To test the scalability of our algorithms, we run two experiments. In the first experiment, we tested the inference time of our system over datasets with different sizes to see the relation between the data volume and the throughput. In the second experiment, we run our system over fpdLUBM1000 dataset with different number of units (mappers and reducers) to see the relation between the processing units and the throughput. Furthermore, in the second experiment, we set the number of mappers to be the same as the number of reducers. Thus a total number of 128 units means launching 64 mappers and 64 reducers.

The results for the first experiment can be found in table 4. From the table, we can see that the throughput increases significantly while the volume increases. The throughput while processing fpdLUBM8000 dataset is 50% higher than the throughput while processing dataset containing 1000 universities. We attribute this performance gain to the platform startup overhead which is amortized over a larger processing time for large datasets. The platform overhead is also responsible for the non-linear speedup in Table 3 which contains the results of the second test. Figure 1 gives a direct illustration of the overhead effect. In Figure 1, if we subtract a constant from the time dimension of each data point, then the time is inversely proportional to the number of units. Since the running time should be inversely proportional to the speed, after eliminating the effect of the platform overhead, the system's performance speeds up linearly to the increase of number of units.

5 Related Work

[10] is the first work to extend RDFS with fuzzy vagueness. In [4], we further propose the fuzzy pD^* semantics which allows some useful OWL vocabularies,

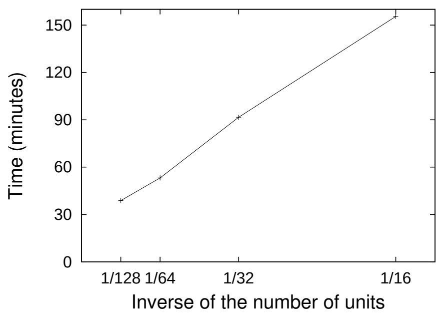

Fig. 1. Time versus inverse of number of mappers

such as TransitiveProperty and SameAs. In [11] and [6], a more general framework called annotated RDF to represent annotation for RDF data and a query language called AnQL were proposed.

As far as we know, this is the first work on applying the MapReduce framework to tackle large scale reasoning in fuzzy OWL. Pan et al. in [7] propose a framework of fuzzy query languages for fuzzy ontologies. However, they mainly concerns the query answering algorithms for these query languages over fuzzy DL-Lite ontologies. Our work concerns the inference problem over large scale fuzzy pD^* ontologies.

We briefly discuss some related work on scalable reasoning in OWL and RDF. None of them takes into account of fuzzy information.

Schlicht and Stuckenschmidt [8] show peer-to-peer reasoning for the expressive ALC logic but focusing on distribution rather than performance. Soma and Prasanna [9] 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 Weaver and Hendler [14], straightforward parallel RDFS reasoning on a cluster is presented. But this approach splits the input to independent partitions. Thus this approach is only applicable for simple logics, e.g. RDFS without extending the RDFS schema, where the input is independent.

Newman et al. [5] 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).

Urbani et al. [13] develop the MapReduce algorithms for materializing RDFS inference results. In [12], they further extend their methods to handle OWL pD^* fragment, and conduct experiment over a dataset containing 100 billion triples.

6 Conclusion

In this paper, we proposed MapReduce algorithms to process forward inference over large scale data using fuzzy pD^* semantics (i.e. an extension of OWL

Horst semantics with fuzzy vagueness). We first identified the major challenges to handle the fuzzy information using the MapReduce framework, and proposed a solution for tackling each of them. Furthermore, we implemented a prototype system for the evaluation purpose. The experimental results show that the running time of our system is comparable with that of WebPIE, the state-of-the-art inference engine for large scale OWL ontologies in pD^* fragment. As a future work, we will apply our system to some applications, such as Genomics and multimedia data management .

7 Acknowledgement

Guilin Qi is partially supported by NSFC (61003157), Jiangsu Science Foundation (BK2010412), and the Key Laboratory of Computer Network and Information Integration (Southeast University).

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of OSDI' 04. pp. 137 – 147 (2004)
2. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics 3(2), 158 – 182 (2005)
3. Horst, H.J.: 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)
4. Liu, C., Qi, G., Wang, H., Yu, Y.: Fuzzy Reasoning over RDF Data using OWL Vocabulary. In: Proc. of WI' 11 (2011)
5. Newman, A., Li, Y.F., Hunter, J.: Scalable semantics - the silver lining of cloud computing. In: Proc. of ESCIENCE 08 (2008)
6. Nuno Lopes, Axel Polleres, U.S., Zimmermann, A.: Anql: Sparqling up annotated rdf. In: Proc. of ISWC' 10. pp. 518 – 533 (2010)
7. Pan, J.Z., Stamou, G., Stoilos, G., Taylor, S., Thomas, E.: Scalable querying services over fuzzy ontologies. In: Proc. of WWW' 08. pp. 575–584 (2008)
8. Schlicht, A., Stuckenschmidt, H.: Peer-to-peer reasoning for interlinked ontologies. vol. 4, pp. 27–58 (2010)
9. Soma, R., Prasanna, V.: Parallel inferencing for owl knowledge bases. In: Proc. of ICPP' 08. pp. 75 –82 (2008)
10. Straccia, U.: A minimal deductive system for general fuzzy RDF. In: Proc. of RR' 09. pp. 166 – 181 (2009)
11. Straccia, U., Lopes, N., Lukacsy, G., Polleres, A.: A general framework for representing and reasoning with annotated semantic web data. In: Proc. of AAAI' 10). pp. 1437 – 1442. AAAI Press (2010)
12. Urbani, J., Kotoulas, S., Maassen, J., Harmelen, F.V., Bal, H.: Owl reasoning with webpie: calculating the closure of 100 billion triples. In: Proc. of ESWC' 10. pp. 213 – 227 (2010)
13. Urbani, J., Kotoulas, S., Oren, E., Harmelen, F.V.: Scalable distributed reasoning using mapreduce. In: Proc. of ISWC' 09. pp. 374 – 389 (2009)
14. Weaver, J., Hendler, J.A.: Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In: Proc. of ISWC' 09. pp. 682–697 (2009)