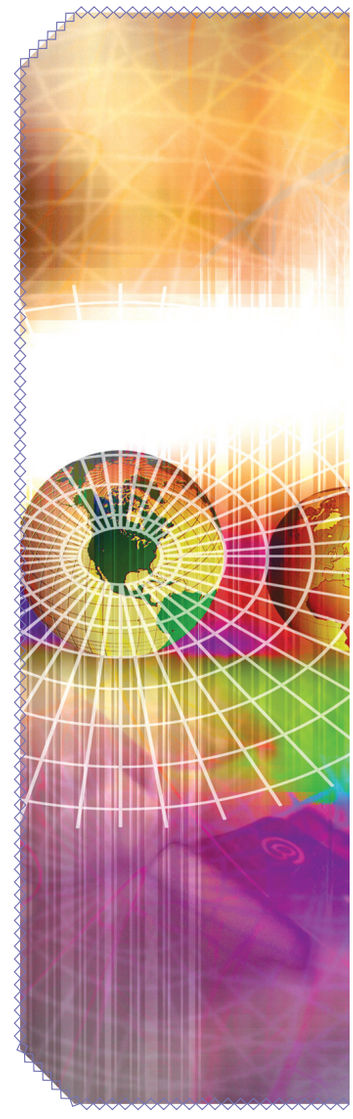
Reasoning with Large Scale Ontologies in Fuzzy *pD** Using MapReduce



Chang Liu, *Shanghai Jiao Tong University, CHINA*

Guilin Qi, *Southeast University, CHINA*

Haofen Wang and Yong Yu, *Shanghai Jiao Tong University, CHINA*

***Abstract*—**The MapReduce framework has proved to be very efficient for data-intensive tasks. Earlier work has successfully applied MapReduce for large scale RDFS/OWL reasoning. In this paper, we move a step forward by considering scalable reasoning on 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 be applied to solve the scalability issue of fuzzy reasoning in OWL. While most of the optimizations considered by the existing MapReduce framework for pD^* semantics are also applicable for fuzzy pD^* semantics, unique challenges arise when we handle the fuzzy information. Key challenges are identified with solution proposed for each of these challenges. Furthermore, a prototype system is implemented 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.

Digital Object Identifier 10.1109/MCI.2012.2188589

Date of publication: 13 April 2012

I. Introduction

The Resource Description Framework (RDF) [1] is one of the major representation standards for the Semantic Web. RDF Schema (RDFS) [2] 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) [4] vocabulary (e.g., owl:sameAs) was introduced, which extends the RDFS semantics to the pD^* fragment of OWL. The OWL pD^* fragment provides a complete set of entailment rules, guaranteeing 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 has been an increasing interest in extending RDF to represent vague information on the Web. Fuzzy RDF allows us to state a triple is true to a certain degree. For example, (*Tom, eat, pizza*) is true with a degree which is at least 0.8. Fuzzy RDFS [5] can handle class hierarchy and property hierarchy with fuzzy degree. However, both Fuzzy RDF and fuzzy RDFS have limited

1556-603X/12/$31.00©2012IEEE

© IMAGESTATE

expressive power to represent information in some real life applications of ontologies, such as biomedicine and multimedia. Therefore, in [6], we extended the OWL pD^* fragment with fuzzy semantics to provide more expressive power than fuzzy RDF(S). However, none of these works focuses on the efficiency issue of the reasoning problem. Since fuzzy RDFS semantics and fuzzy pD^* semantics are focussed on handling large scale semantic data, it is critical to provide a scalable reasoning algorithm for them.

Earlier works (e.g. [7] and [8]) have proved that MapReduce [9] is a very efficient framework to handle the computation of the *closure* of a RDF graph containing up to 100 billion triples under RDFS and pD^* semantics. As such, MapReduce may be helpful to scalable reasoning in fuzzy RDFS semantics and fuzzy pD^* semantics. It turns out that this is a non-trivial problem as the computation of the closure under fuzzy RDFS and 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 in the closure of a fuzzy RDFS (or fuzzy pD^*) graph. Although most of the optimizations considered by the existing MapReduce framework for pD^* semantics are also applicable for fuzzy pD^* semantics, unique challenges arise when we handle the fuzzy information. For example, [8] proposed an algorithm to handle `TransitiveProperty`. In the fuzzy case, however, we will show in Section III-D3 that calculating the `TransitiveProperty` closure is essentially a variation of the all-pairs shortest path calculation problem, and the algorithm proposed in [8] cannot deal with this problem. Instead efficient algorithms are required to deal with this problem.

In this paper, the proposal is to to build a scalable and efficient reasoning engine for computing closure of a fuzzy RDF graph under fuzzy pD^* semantics. Firstly, a reasoning algorithm is recommended for fuzzy pD^* semantics based on the MapReduce framework. Thereafter MapReduce algorithms is presented for fuzzy pD^* rules together with some novel optimizations. Finally, a prototype system is implemented to

TABLE 1 Fuzzy RDFS entailment rules.

CONDITION	CONCLUSION	
F-RDFS1	$(v, p, l)[n]$	$(b_l,$ `type, Literal`$)$ [1]
F-RDFS1X	$(b_l,$ `type, Literal`$)[n]$	$(b_l,$ `type, Literal`$)$ [1]
F-RDFS2	$(p,$ `domain`$, u)[n]$ $(v, p, w)[m]$	$(v,$ `type`$, u)[n \otimes m]$
F-RDFS3	$(p,$ `range`$, w)[n]$ $(v, p, w)[m]$	$(v,$ `type`$, w)[n \otimes m]$
F-RDFS4A	$(v, p, w)[n]$	$(v,$ `type, Resource`$)[1]$
F-RDFS4B	$(v, p, w)[n]$	$(w,$ `type, Resource`$)[1]$
F-RDFS5	$(v,$ `subPropertyOf`$, w)[n]$ $(w,$ `subPropertyOf`$, u)[m]$	$(v,$ `subPropertyOf`$, u)[n \otimes m]$
F-RDFS6	$(v,$ `type, Property`$)[1]$	$(v,$ `subPropertyOf`$, v)[1]$
F-RDFS7X	$(p,$ `subPropertyOf`$, q)[n]$ $(v, p, w)[m]$	$(v, q, w)[n \otimes m]$
F-RDFS8	$(v,$ `type, Class`$)[n]$	$(v,$ `subClassOf, Resource`$)[n]$
F-RDFS9	$(v,$ `subClassOf`$, w)[n]$ $(u,$ `type`$, v)[m]$	$(u,$ `type`$, w)[n \otimes m]$
F-RDFS10	$(v,$ `type, Class`$)[n]$	$(v,$ `subClassOf`$, v)[1]$
F-RDFS11	$(v,$ `subClassOf`$, w)[n]$ $(w,$ `subClassOf`$, u)[m]$	$(v,$ `subClassOf`$, u)[n \otimes m]$
F-RDFS12	$(v,$ `type, ContainerMembershipProperty`$)[n]$	$(v,$ `subPropertyOf, member`$)[n]$
F-RDFS13	$(v,$ `type, Datatype`$)[n]$	$(v,$ `subClassOf, Literal`$)[1]$

evaluate all these optimizations, and conduct extensive experiments. The experimental results also show that the running time of this proposed system is comparable with that of WebPIE [8], the state-of-the-art inference engine for scalable reasoning in pD^* semantics. This paper is extended and revised from our conference paper [10].

The rest of the paper is organized as follows. In Section II, we introduce the background knowledge about fuzzy pD^* reasoning and the MapReduce framework. Then we propose our MapReduce algorithm for fuzzy pD^* reasoning in Section III. The experiment results are provided in Section IV. Discussions of related work can be found in Section V. Finally we conclude this paper in Section VI.

II. Background Knowledge

In this section, the fuzzy pD^* entailment rule set is introduced in Section II-B, followed by an explanation of the MapReduce framework for reasoning in OWL pD^* fragment in Section II-C.

A. RDF, RDFS and pD* Semantics

An RDF [1] graph is defined as a subset of the set $\mathbf{UB} \times \mathbf{U} \times \mathbf{UBL}$, where **U**, **B** and **L** represent the set of *URI references, Blank nodes*, and *Literals respectively*. The element (s, p, o) of an RDF graph is a *triple*. Intuitively, we use a triple to represent a statement. For example, we can use a triple $(Tom, eat, pizza)$ to state that Tom eats pizza. RDF properties may be considered as relationships between resources. RDF, however, provides no mechanisms for describing the relationships between these properties and other resources. RDF Schema (RDFS) [2], which plays the role of the RDF vocabulary description language, defines classes and properties that may be used to describe classes, properties and other resources.

However, RDF and RDFS only provide a very limited expressiveness. In [3], a subset of Ontology Web Language (OWL) [4] 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). The OWL vocabulary supported by pD^* semantics include FunctionalProperty, InverseFunctionalProperty, SymmetricProperty, TransitiveProperty, sameAs, inverseOf, equivalentClass, equivalentProperty, hasValueOf, someValuesFrom and allValuesFrom.

B. Fuzzy RDFS and pD* Reasoning

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

Fuzzy RDFS semantics extends RDFS semantics with fuzzy semantics. The complete and sound entailment rule set of fuzzy RDFS is listed in Table 1. However, fuzzy RDFS has limited expressive power to represent information in some real life applications of ontologies, such as biomedicine and multimedia. Therefore, fuzzy pD^* semantics was proposed in [6] to extends pD^* semantics with fuzzy semantics. To handle the additional vocabularies, we use the fuzzy P-entailment rule set listed in Table 2 along with the fuzzy RDFS rules. The notion of a (partial) closure can be easily extended to the fuzzy case.

One of the key notions in both fuzzy RDFS and 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 under fuzzy RDFS semantics is defined to be the largest fuzzy degree n such that $t[n]$ can be derived from G by applying the fuzzy RDFS-entailment rules, or 0 if no such fuzzy triple can be derived. The definition of BDB of a triple under fuzzy pD^* semantics can be similarly defined.

C. MapReduce Framework

MapReduce is a programming model introduced by Google for large scale data processing [9]. 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.

To illustrate the key factor affecting the performance of the MapReduce program, we briefly introduce the implementation of Hadoop[1] which is an open source MapReduce

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

implementation. When a MapReduce program is submitted to the Hadoop job manager, the job manager will first split the input data into several pieces. Then a mapper will be created for each input split, and assigned to an empty map slot (machine). The mappers will generate a lot of key/value pairs. Then the job manager will assign a reducer to each empty reduce slot, and each key/value pair that is outputted by any mapper will be transmitted to the corresponding reducer. When all mappers complete and the data are transferred to the reducers' side, each reducer will first shuffle all the data so that all the key/value pairs with the same key can be grouped in order to be executed together. Then the reducer will process each different key and a list of values at once, and generate the output.

If there is a mapper that is very slow, then the whole program will have to wait for its mapper slot, while other mapper slots which have been completed earlier have to remain idle and cannot be assigned to other mappers. This situation is called skew.

According to these implementations, the following factors will influence the performance of the MapReduce program. They also give important principles to develop efficient MapReduce program.

- On the mapper side, only when all the mappers complete their jobs, then the reducer will start to work. If there is a mapper that is very slow, then the whole program will have to wait for its mapper slot, while other mapper slots which have been completed earlier have to remain idle and cannot be assigned to other mappers. This situation is called *skew*. Since the run time of a mapper is typically linear to the amount of data it is assigned to, this kind of skew is usually caused by unbalanced data partition. In order to improve the performance, we should split the input data as equally as possible.
- Before the reducer is executed, all key/value pairs will be transmitted to the same machine, and a shuffling process must be executed first in order to group key/value pairs with the same key together. This process is the major cost of each MapReduce program. However, for many tasks, the processing algorithm only needs to be executed on the data set in parallel. Thus, the reducers are not needed for these tasks. In this case, the MapReduce framework allows the program to contain only the map side. Thus, the costly data transmission and shuffling process can be avoided to improve the performance.
- On the reducer side, there is also a skew problem. The reducer processing a popular key will run very slowly. Therefore, the mappers' output keys should be carefully designed to ensure that the sizes of all partitions is relatively balanced.
- There is no restriction on how many values there are for the same key. We usually cannot assume that all values sharing a same key can be loaded into the memory. If we force the reducer to load all such values into the memory, in the worst case, it will result in an OutOfMemory error. Thus the reducer should operate on a stream of values instead of a set of values.

III. MapReduce Algorithms For Fuzzy *pD** Reasoning

In this section, we first illustrate how a MapReduce program can be used to apply a fuzzy rule. Then, we give an overview of the challenges when applying the MapReduce framework to

TABLE 2 Fuzzy *P*-entailment rules.

CONDITION	CONCLUSION	
F-RDFP1	(*p*, type, FunctionalProperty) [*n*] (*u*, *p*, *v*) [*m*] (*u*, *p*, *w*) [*l*]	(*v*, sameAs, *w*)[*l*⊗*m*⊗*n*]
F-RDFP2	(*p*, type, InverseFunctionalProperty) [*n*] (*u*, *p*, *w*)[*m*] (*v*, *p*, *w*)[*l*]	(*u*, sameA, *v*)[*l*⊗*m*⊗*n*]
F-RDFP3	(*p*, type, SymmetricProperty)[*n*] (*v*, *p*, *w*)[*m*]	(*w*, *p*, *v*)[*n*⊗*m*]
F-RDFP4	(*p*, type, TransitiveProperty)[*n*] (*u*, *p*, *v*)[*m*] (*v*, *p*, *w*)[*l*]	(*u*, *p*, *w*)[*n*⊗*m*⊗*l*]
F-RDFP5(AB)	(*v*, *p*, *w*)[*n*]	(*v*, sameAs, *v*)[1], (*w*, sameAs, *w*)[1]
F-RDFP6	(*v*, sameAs, *w*)[*n*]	(*w*, sameAs, *v*)[*n*]
F-RDFP7	(*u*, sameAs, *v*)[*n*] (*v*, sameAs, *w*)[*m*]	(*u*, sameAs, *w*)[*n*⊗*m*]
F-RDFP8AX	(*p*, inverseOf, *q*)[*n*] (*v*, *p*, *w*)[*m*]	(*w*, *q*, *v*)[*n*⊗*m*]
F-RDFP8BX	(*p*,inverseOf, *q*)[*n*] (*v*, *q*, *w*)[*m*]	(*w*, *p*, *v*)[*n*⊗*m*]
F-RDFP9	(*v*, type, Class)[*n*] (*v*, sameAs, *w*)[*m*]	(*v*, subClassOf, *w*)[*m*]
F-RDFP10	(*p*, type, Property)[1] (*p*, sameAs, *q*)[*m*]	(*p*, subPropertyOf, *q*)[*m*]
F-RDFP11	(*u*, *p*, *v*)[*n*] (*u*, sameAs, *u*´)[*m*] (*v*, sameAs, *v*´)[*l*]	(*u*´, *p*, *v*´)[*n*⊗*m*⊗*l*]
F-RDFP12(AB)	(*v*, equivalentClass, *w*)[*n*] ⇒(*v*, subClassOf, *w*)[*n*], (*w*, subClassOf, *w*)[*n*]	
F-RDFP12C	(*v*, subClassOf, *w*)[*n*] (*w*, subClassOf, *v*)[*m*]	(*v*, equivalentClass, *w*)[min(*n*, *m*)]
F-RDFP13(AB)	(*v*, equivalentProperty, *w*)[*n*]⇒(*v*, subPropertyOf, *w*)[*n*],	(*w*, subPropertyOf, *w*)[*n*]
F-RDFP13C	(*v*, subPropertyOf, *w*)[*n*] (*w*, subPropertyOf, *v*)[*m*]	(*v*, equivalentClass, *w*)[min(*n*, *m*)]
F-RDFP14A	(*v*, hasValueOf, *w*)[*n*] (*v*, onProperty, *p*)[*m*] (*u*, *p*, *w*)[*l*]	(*u*, type, *v*)[*n*⊗*m*⊗*l*]
F-RDFP14BX	(*v*, hasValueOf, *w*)[*n*] (*v*, onProperty, *p*)[*m*] (*u*, type, *v*)[*l*]	(*u*, *p*, *w*)[*n*⊗*m*⊗*l*]
F-RDFP15	(*v*, someValueFrom, *w*)[*n*] (*v*, onProperty, *p*)[*m*] (*u*, *p*, *x*)[*l*] (*x*, type, *w*)[*k*]	(*u*, type, *v*)[*n*⊗*m*⊗*l*⊗*k*]
F-RDFP16	(*v*, allValuesfrom, *w*)[*m*] (*v*, onProperty, *p*)[*n*] (*u*, type, *v*)[*l*] (*u*, *p*, *x*)[*k*]	(*x*, type, *w*)[*n*⊗*m*⊗*l*⊗*k*]

ALGORITHM 1: Map function for rule f-rdfs2.

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

fuzzy pD^* reasoning. Finally, we present our solutions to handle these challenges.

A. Naive MapReduce Algorithms 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.

ALGORITHM 2: Reduce function for rule f-rdfs2.

```
Input: key, iterator values
1: unSet.clear( );
2: vmSet.clear( );
3: for value ∈ values do
4: if value.flag == 'L' then
5: unSet.update(value.u, value.n);
6: else
7: vmSet.update(value.v, value.m);
8: end if
9: end for
10: for i ∈ unSet do
11: for j ∈ vmSet do
12: emit(null, new FuzzyTriple(i.u, 'type', j.v, i.n⊗j.m));
13: end for
14: end for
```

B. Challenges

Even though the fuzzy pD^* entailment rules are quite similar to the pD^* rules, several difficulties arose when we calculated the BDB for each triple by applying the MapReduce framework. The summary of these challenges are as follows:

1) Efficient Implementation

The Naive implementation will encounter some problems and violate the principles to develop efficient MapReduce program introduced in Section II-C. For example, the program for rule f-rdfs2 shown in the last subsection introduces an unnecessary shuffling process and treats the values in the *reduce* function as a set instead of a stream. We will discuss a solution to tackle this problem in Section III-C2 and Section III-D2.

2) Ordering the Rule Applications

In fuzzy pD^* semantics, the reasoner might produce a duplicated triple with different fuzzy degrees before the BDB of such a triple is derived. For example, if the data set contains a fuzzy triple $t[m]$, when a fuzzy triple $t[n]$ with $n > m$ is derived, a duplicated triple is generated. In this case, we should employ a deletion program to reproduce the data set to ensure that for any triple, among all its extended fuzzy triples that can be derived from the MapReduce algorithms, only the one with maximal degrees are kept in the data set. Different orders of rule applications will result in different number of such duplicated triples. To achieve the best performance, we should choose a proper order to reduce the number of duplicated triples. 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). A solution for this problem will be discussed in Section III-C1 and section III-D1.

3) Shortest Path Calculation

In fuzzy pD^* semantics, there are three rules, i.e., f-rdfs5 (subproperty), f-rdfs11 (subclass) and f-rdfp4 (transitive property), that require iterative calculations. 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. We will discuss rules f-rdfs5 and f-rdfs11 in section III-C3 and discuss rule f-rdfp4 in section III-D3.

4) SameAs Rule

The most expensive calculation happens when we process the `sameAs` rules f-rdfp 5(ab), 6, 7, 9, 10 and 11. Fully applying all these rules will result in an extremely large fuzzy triple set, which is unnecessary for the application purpose. In Section III-D4 and Section III-D5, we will discuss how to handle these rules efficiently.

C. Algorithms for Handling Fuzzy RDFS Rules

In this subsection, we present MapReduce algorithms for handling the RDFS rules (rules f-rdfs1 to f-rdfs13). Most of these rules have only one fuzzy triple as a condition so that they can function on its own. These rules are easy to handle. Thus we only considered rules with at least two fuzzy triples in its condition, i.e., rules f-rdfs 2, 3, 5, 7x, 9, 11.

Generally speaking, the reasoning engine should iteratively perform the rules to see if a fixpoint is reached. However, after carefully looking at these fuzzy RDFS rules, we find that it is possible to order the rules so that every rule can be executed only once to generate all conclusions.

1) Ordering Rules

Generally speaking, the reasoning engine should iteratively perform the rules to see if a fixpoint is reached. However, after carefully looking at these fuzzy RDFS rules, we find that it is possible to order the rules so that every rule can be executed only once to generate all conclusions. We have the following four phases of rule application.

In the first phase, rules f-rdfs 5, 6, and 7 are executed to produce the whole property hierarchy. In the second phase, the domain and range rules (f-rdfs2 and f-rdfs3) will be performed. In the third phase, we execute rules f-rdfs9, 10 and 11 to produce the whole class hierarchy. Finally, we apply the remaining rules.

In the first and the third phase, the reasoning engine will need to recursively call a MapReduce algorithm to calculate the complete hierarchy. This will greatly influence its efficiency. This problem will be discussed in the following subsections. As we will show, in each phase, the reasoning engine only needs to call one MapReduce program. Some duplicated triples may be derived after applying rules. After executing all the four phases, we should launch a program to remove them to guarantee that each triple appears in the closure only once.

2) Loading Schema Triples into the Memory

As shown in section III-A, a join can be handled by a MapReduce program. However, there are two drawbacks in this MapReduce program. Firstly, there is a very expensive shuffling process before the reducers are launched. Secondly, in each reducer, we have to load all values of key/value pairs generated by mappers with the same key into the memory. As we have discussed in section II-C, if there are too many values, the reducer can easily fail.

We find that in all these rules, there is a fuzzy triple in the condition that is a schema triple. In practice, we can assume that the number of schema triples is relatively small. Therefore, we can load them into the memory, and perform the join in the mappers so that we can avoid the above two drawbacks.

We illustrate this idea by considering rule f-rdfs2. The *map* function is provided in Algorithm 3. Before each mapper starts to process the data, it will load all triples in the form of $(p, \texttt{domain}, w)[n]$, and store them in *domainProperty* which is a Map (in our implementation, we use a hash table) that maps each p to the list of (w, n) pairs. Then the MapReduce program only has to parallelly enumerate every fuzzy triple $(u, p, v)[m]$to output the fuzzy triples $(u, \texttt{type}, w)[n \otimes m]$. Thus we do not need a *reduce* program.

3) Calculating the `subClassOf` and the `subPropertyOf` Closure

Rules f-rdfs5, 6, and 7, and rules f-rdfp12(abc) are relevant to the `subPropertyOf` property while rules f-rdfs9, 10, and 11, and rules f-rdfp13(abc) are relevant to the `subClassOf` property. We only discuss rules that are relevant to `subClassOf` as rules that are relevant to the `subPropertyOf` property can be handled similarly. 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).

We call the triples in the form of $(u, \texttt{subClassOf}, v)[n]$ to be `subClassOf` triples. It is easy to see that rule f-rdfs11 needs to recursively calculate the transitive closure of `subClassOf` triples. However, since they are schema triples, as we have discussed previously, we can load them into the memory so that we can avoid the reiterative execution of the MapReduce program of rule f-rdfs11. In fact, 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 $(u, \texttt{subClassOf}, v)[n]$ that can be derived from the original fuzzy RDF graph by only applying rule f-rdfs11, there must be a chain of classes $w_0 = u, w_1, \ldots, w_k = v$ and a list of fuzzy degrees $d_1, \ldots, d_k$ such

ALGORITHM 3: Map function for rule f-rdfs2 (loading schema triples).

```
Input: key, triple
1: if domainProperty.containsKey(triple.predicate) then
2:   for (u,v) ∈ domainProperty.get(triple.predicate) do
3:     emit(new FuzzyTriple(triple.subject, type, u, n⊗triple.
         degree));
4:   end for
5: end if
```

ALGORITHM 4: Calculate the `subClassOf` closure.

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

that for every $i = 1, 2, \ldots, k$, $(w_{i-1}, \texttt{subClassOf}, w_i)[d_k]$ is in the original fuzzy graph and $n = d_1 \otimes d_2 \otimes \ldots \otimes d_k$.

This property is easy to prove by induction.[2] Indeed, from fuzzy triples $(w_0, \textbf{subClassOf}, w_i)[d_1 \otimes \ldots \otimes d_i]$ and $(w_i, \textbf{subClassOf}, w_k)[d_{i+1} \otimes \ldots \otimes d_k]$, we can derive $(w_0, \textbf{SubClassOf}, w_k)[d_1 \otimes \ldots \otimes d_k]$ using f-rdfs11.

So we can use the Floyd-Warshall 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 $(i, \texttt{subClassOf}, j)$. The algorithm iteratively updates 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.

ALGORITHM 5: 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
```

[2] The full proof can be found in our technique report which is available at http://apex.sjtu.edu.cn/apex_wiki/fuzzypd.

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

For rule f-rdfp12(abc), since `equivalentClass` triples are also schema triples, we load them into the memory and combine them into the `subClassOf` graph. Specifically, when we load a triple $(i, \texttt{equivalentClass}, j)[n]$ into the memory, *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 $n = \min(w(i, j), w(j, i)) > 0$.

D. Algorithms for Fuzzy pD Rules*

For fuzzy pD^* entailment rules, there is no ordering that can avoid iterative execution. Thus we have to iteratively apply the rules until a fixpoint is reached. In the following, we first give the main reasoning algorithm, we then discuss the optimizations for fuzzy pD^* rules.

1) Overview of the Reasoning Algorithm

Our main reasoning algorithm is Algorithm 5, 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-entailment rules (from rdfp1 to rdfp16). Since some fuzzy P-entailment 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 fixpoint is reached (line 4 to line 6).

For the fuzzy P-entailment rules, there is no way to avoid a fixpoint iteration. So we employ an iterative algorithm to calculate the closure under P-entailment rules. 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 `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. In the first and last phases, there are rules that require more than one join, which turns out to be very inefficient. We first discuss how to avoid these multiple joins in Section III-D2. We then discuss the solution to deal with transitive property in Section III-D3. Finally the solution to tackle `sameAs` rules will be discussed in Section III-D4 and Section III-D5.

2) Multiple Joins

For fuzzy P-entailment rules that have more than two triples in their condition, more than one join should be performed in

order to derive the result of such a rule. These rules are f-rdfp 1, 2, 4, 11, 14a, 14bx, 15 and 16. Rule f-rdfp4 and f-rdfp11 will be discussed in the next two subsections, thus we only consider the remaining six rules. After carefully examining these rules, we find that most of the fuzzy triples in the condition part of these rules are schema triples, and the number of assertion triples in each single rule is at most two. Therefore, we can leverage the technique discussed in the last section, i.e., loading schema triples into the memory.

We illustrate this idea by considering rule f-rdfp15:

$$(v, \texttt{someValueFrom}, w)[n], (v, \texttt{onProperty}, p)[m], (u, p, x)[l], (x, \texttt{type}, w)[k] \Rightarrow (u, \texttt{type}, v)[n \otimes m \otimes l \otimes k].$$

In the condition of this rule, $(v, \texttt{someValueFrom}, w)[n]$ and $(v, \texttt{onProperty}, p)[m]$ are schema triples. Thus, before the mappers are launched, they are loaded into the memory and stored in two Maps, e.g. two hash tables: someValuesFrom maps each w to a list of (v, n) pairs, and onProperty maps each p to a list of (v, m) pairs. The *map* function is given in Algorithm 6. In the mapper, we perform two joins. Lines 1 to 5 in Algorithm 6 calculate the join over $(v, \texttt{someValueFrom}, w)[n]$and $(x, \texttt{type}, w)[k]$, resulting in a table with schema $(x, v, n \otimes k)$, and lines 6 to 10 calculate the join over $(v, \texttt{onProperty}, p)[m]$ and $(u, p, x)[l]$, resulting in a table with schema $(x, v, u, m \otimes l)$. Then the mappers output these two tables using (x, v) as the key so that the *reduce* function which is given in Algorithm 7 can perform the final join over these two tables.

3) 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 $u_0 = a, u_1, \ldots, u_k = b$ and a list of fuzzy degrees $d_1, \ldots, d_k$ such that $m = d_1 \otimes n \otimes d_2 \otimes \ldots \otimes n \otimes d_k$, and for every $i = 1, 2, \ldots, k$, $(u_{i-1}, p, u_p)[d_i]$ is in the original fuzzy RDF graph. Furthermore, in one of such chains, $u_i \neq u_j$, if $i \neq j$and $i, j \geq 1$.

This property is easy to prove by induction.[3] Indeed, from fuzzy triples $(u_0, p, u_i)[d_1 \otimes \ldots \otimes d_i \otimes n^i]$, $(u_i, p, w_k)[d_{i+1} \otimes \ldots \otimes d_k \otimes n^{k-i}]$ and $(p, \texttt{type}, \texttt{TransitiveProperty})[n]$, we can derive $(u_0, p, w_k)[d_1 \otimes \ldots \otimes d_k \otimes n^{k+1}]$ using f-rdfp4. We use an iterative algorithm to calculate this transitive closure. In each iteration, we execute a MapReduce program using Algorithm 8 as the *map* function and Algorithm 9 as the *reduce* function.

We use `getTransitiveDegree(p)` function to get the maximal *degree* such that $(p, \texttt{type}, \texttt{TransitiveProperty})[degree]$ is in the graph. Since these triples are schema triples, we can load them into the memory before the mappers and reducers are executed. Suppose l is the number of iterations that the transitive closure calculation algorithm already executes. In the *map* function, l is required as input. For any fuzzy triple $(a, p, b)[m]$, there is at least one chain $u_0 = a, u_1, \ldots, u_k = b$ 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.

ALGORITHM 6: Map function for rule f-rdfp15.

```
Input: key, triple
1:  if triple.predicate == 'type' and sameValuesFrom.
    containsKey(triple.object) then
2:    for (v,n) ∈ sameValuesFrom.get(triple.ojbect) do
3:      emit({triple.subject, v}, {flag='L', degree=n ⊗triple.
        degree});
4:    end for
5:  end if
6:  if onProperty.containsKey(triple.predicate) then
7:    for (v,m) ∈ onProperty.get(triple.predicate) do
8:      emit({triple.object, v}, {flag='R', u=triple.subject,
        degree= m ⊗triple.degree});
9:    end for
10: end if
```

ALGORITHM 7: Reduce function for rule f-rdfp15.

```
Input: key, iterator values
1:  nmax = 0;
2:  mSet.clear( );
3:  for value ∈ values do
4:    if value.flag == 'L' then
5:      nmax = value.degree > nmax ? value.degree: nmax;
6:    else
7:      mSet.update(value.u, value.m);
8:    end if
9:  end for
10: for (u,m) ∈ mSet do
11:   emit(null, new FuzzyTriple(u, 'type', key.v, nmax⊗m));
12: end for
```

[3] The full proof can be found in our technique report which is available at http://apex.sjtu.edu.cn/apex_wiki/fuzzypd.

ALGORITHM 8: Map function for rule f-rdfp4.

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

If $(a,p,b)[m]$ has a chain $u_0,u_1,\ldots,u_k$ with length k, and it can be derived from $(a,p,t)[m_1]$ and $(t,p,b)[m_2]$ in the l-th iteration, then we have $m_1+m_2=m$, $m_1=2^{l-2}$ or 2^{l-1}, and $2^{l-2}<m_2\le 2^{l-1}$. We can show that the integer equation $m_1+m_2=m$ has a unique solution satisfying $m_1=2^{l-2}$ or $m_1=2^{l-1}$, and $2^{l-2}<m_2\le 2^{l-1}$. 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 $2^{l-1}<length\le 2^l$, it will be derived within l iterations. As a consequence, the algorithm will terminate within $\log N$ iterations where N is the number of all instances in the graph.

ALGORITHM 9: 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, new FuzzyTriple(i.subject, key.predicate,
        j.object,
        i.degree ⊗ j.degree ⊗ getTransitiveDegree
        (key.predicate)));
    end for
  end for
```

4) Handling Certain `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:

$$(u,p,v)[n],(u,\mathbf{sameAs},u')[m],(v,\mathbf{sameAs},v')[l],$$
$$(p,\mathbf{sameAs},p')[k]\Rightarrow(u',p',v')[n\otimes m\otimes l\otimes k].$$

For convenience, we call a fuzzy triple in the form of $(i,\mathbf{sameAs},j)[n]$ a `sameAs` triple. We further call the `sameAs` triples with fuzzy degree 1 the *certain* `sameAs` *triples*, and those with fuzzy degrees less than 1 the *vague* `sameAs` *triples*.

For certain `sameAs` triples, we can employ a technique introduced in [8], called *canonical representation*, to improve the performance. In some real applications, such as the Linking Open Data project, most of the `sameAs` triples are certain when they are used to link different URIs across different datasets. Thus this technique will be useful in practice.

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 by its unique key results in a more compact representation of the RDF graph without loss of completeness of inference.

To achieve this goal, the algorithm can be divided into two phases. In the first phase, we should compute the canonical representation of each instance. In the second phase, we should replace each instance by its canonical representation.

In the first phase, we choose the canonical representation of each instance to be the connected instance with minimal ID. Here, we say instance i is connected with instance j if there is a chain of $i_0,i_1,\ldots,i_k$ where $i_0=i$, $i_k=j$ and $(i_t,\mathbf{sameAs},i_{t+1})[1]$ or $(i_{t+1},\mathbf{sameAs},i_t)[1]$ belongs to the graph for $t=0,1,\ldots,k-1$. This minimal ID is computed by an algorithm which iteratively call a MapReduce program. After the k-th iteration, the triple set contains all certain sameAs triples $(i,\mathbf{sameAs},j)[1.0]$ where i is the instance with minimal ID that is connected with j in the `sameAs` graph and the distance between i and j is at most 2^k. Suppose there are n instances in the `sameAs` graph, then after $\log n$ iterations, the algorithm will find the canonical representations of all instances.

The *map* function and the *reduce* function are given in Algorithm 10 and Algorithm 11 respectively. The mapper processes only certain `sameAs` triples, and after each iteration, the input certain `sameAs` triples will be deleted from the fuzzy triple set. According to our assumption, at the beginning of the k-th iteration, the mapper will only process fuzzy triples in the form of $(u,\mathbf{sameAs},v)[1.0]$ where u is the instance with minimal ID

which is connected with v and the distance between u and v is at most 2^{k-1}. When it scans a triple in the form of $(u, \mathbf{sameAs}, v)[1.0]$, the mapper will emit both (u, v) and (v, u). Then for each *key*, the reducer will process all values v which is connect with *key* within 2^{k-1} distance. The reducer chooses the canonical representation *min* to be the minimal number in the set containing all values v and the key, and emit a fuzzy triple $(min, \mathbf{sameAs}, v)[1.0]$. It is easy to see that *min* should be the minimal ID corresponding to an instance i that is connected to v where the distance between i and v is at most 2^k through instance with ID *key*. Therefore, our algorithm will correctly compute the canonical representation after $\log n$ iterations.

After the first phase, we will compute a canonical representation table which stores $(i, \mathbf{sameAs}, CR(i))[1]$ where $CR(i)$ is the canonical representation of i for each instance i. In the second phase, we shall perform a join between this canonical representation table and the whole fuzzy triple set. This join can be simply calculated using a MapReduce program.

5) Handling Vague `SameAs` Closure

However, not all `sameAs` triples are certain `sameAs` triples. The fuzzy pD^* semantics allows using `sameAs` triples to represent the similarity information. In this case, 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:

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

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 under 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:

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

From this graph, we can derive the fuzzy triple $(v, \mathbf{type}, r)[0.1]$, and this is a fuzzy BDB triple from G', which means we cannot derive the fuzzy triple $(v, \mathbf{type}, r)[0.8]$. 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. For this reason, we must store all of these vague sameAs triples and calculate the `sameAs` closure using rules f-rdfp6 and f-rdfp7 to ensure the inference is complete.

Materializing the result of applying 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.

ALGORITHM 10: Map function to calculate the canonical representation.

```
Input: key, triple
1: emit(triple.subject, triple.object);
2: emit(triple.object, triple.subject);
```

In the following, we use rule f-rdfs2 mentioned in III-A 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:

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

We use Algorithm 12 to replace Algorithm 1 as the *map* function. The difference is that Algorithm 12 uses a loop between line 2 and line 5. In practice, vague `sameAs` triples are relatively few. Thus we can load them into the memory and compute the `sameAs` closure before the mappers are launched. When the mapper scans a triple in the form of $(p, \mathbf{domain}, u)[m]$, it 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 output by the mapper in line 7. Thus the reducer will generate fuzzy triples in the form of $(v, \mathbf{type}, u)[n \otimes m \otimes k]$ as desired.

Finally, we discuss the problem of handling `sameAs` triples while processing rules f-rdfp1 and f-rdfp2. We only discuss rule f-rdfp1 since the other can be handled similarly. Consider a fuzzy graph G containing the following $n+1$ fuzzy triples:

ALGORITHM 11: Reduce function to calculate the canonical representation.

```
Input: key, iterator values
1:  min = key;
2:  for v ∈ values do
3:    if v < min then
4:      min = v;
5:    end if
6:  end for
7:  for v ∈ values do
8:    emit(null, new FuzzyTriple(min, sameAs, v, 1.0));
9:  end for
```

The computation of the transitive closure by applying rule f-rdfp4 is essentially calculating the all-pairs shortest path on the instance graph.

ALGORITHM 12: Map function for rules f-rdfs2 and f-rdfp11.

Input: key, triple

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

$$(a, p, b_1)\ [m_1]\ \ (a, p, b_2)[m_2]\ \ldots\ (a, p, b_n)\ [m_n]$$
$$(p, \texttt{type}, \texttt{FunctionalProperty})[k]$$

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

IV. Experiment

We implemented a prototype system, called FuzzyPD, based on the Hadoop framework[4], which is an open-source Java implementation of MapReduce. Hadoop uses a distributed file system, called HDFS[5] to manage executions details such as data transfer, job scheduling, and error management.

Since there is no system supporting fuzzy pD^* reasoning, we ran our system over the standard LUBM data, and validated it against the WebPIE reasoner to check the correctness of our algorithms. Our system can produce the same results as WebPIE does.

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.

A. Datasets

Since there is no real fuzzy RDF data available, we generated fuzzy degrees for triples in some crisp benchmark ontologies, i.e., DBPedia [11] core ontology and LUBM ontologies [12]. For DBPedia core ontology, we assigned a randomly chosen fuzzy degree to each triple. For LUBM ontologies, we extended the fuzzy LUBM dataset (called fLUBM dataset) generated for fuzzy DL-Lite semantics in [13]. The fLUBM dataset adds two fuzzy classes, called `Busy` and `Famous`. The fuzzy degrees of triples stating an individual belong to these two fuzzy classes are generated according to the number of courses taught or taken by the individual, and the number of 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 added a fuzzy triple $(i,\ \texttt{type},\ \texttt{LessBusy})\ [b_i/0.5]$ into the dataset; if $0.5 \le b_i < 0.7$, we generated a fuzzy triple $(i,\ \texttt{type},\ \texttt{NormalBusy})\ [b_i/0.7]$; otherwise, we generated a fuzzy triple $(i,\ \texttt{type},\ \texttt{VeryBusy})\ [b]$. We added two fuzzy triples, $(\texttt{LessBusy},\ \texttt{subClassOf},\ \texttt{Busy})\ [0.5]$ and $(\texttt{VeryBusy},\ \texttt{subClassOf},\ \texttt{Busy})\ [1.0]$ 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 $(i,\ \texttt{youngerThan},\ j)\ [age_i/age_j]$ 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 $(f,\ \texttt{emailAddress},\ e)\ [d]$ 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 d to be 0.9 with probability 1%, so that a small set of vague `sameas` triples can be generated. Similarly, we can generate other emailAddress related triples for the master and doctoral students similarly. We call the new dataset as fpdLUBM dataset[6].

B. Experimental Results

1) Fuzzy RDFS Reasoning

Since we employ the fuzzy RDFS reasoning algorithm as a subroutine of the fuzzy pD^* reasoning algorithm, the running time of

[4] http://hadoop.apache.org/
[5] http://hadoop.apache.org/hdfs/

[6] The dataset is available at http://apex.sjtu.edu.cn/apex_wiki/fuzzypd.

the former algorithm is definitely smaller than that of the later. In this subsection, we only study the relation between the performance of our algorithm and the number of computing units (mappers and reducers).

TABLE 3 Scalability for fuzzy RDFS reasoning.

NUMBER OF UNITS	128	64	32	16	8	4	2
TIME (SECONDS)	122.653	136.861	146.393	170.859	282.802	446.917	822.269
SPEEDUP	6.70	6.01	5.62	4.81	2.91	1.84	1.00

We used generated fuzzy version of the DBPedia core ontology which contains 26996983 fuzzy triples. After performing fuzzy RDFS reasoning algorithm, 133656 new fuzzy triples were derived. The running time results are listed in Table 3. The result shows that the running time speedup increases along with the number of computing units used. However, this speedup increase is not as linear as expected. The reason is that there is a warmup overhead of the Hadoop system which is unavoidable no matter how many computing units we used. Furthermore, the running time is relatively smaller than this overhead, thus it is hard to calculate the exact running time from the data we measured. Later in Section IV-B3, we will see the results conducted on larger datasets showing a good linear speedup as expected.

2) Comparison with WebPIE

We compared the performance of our system with that of the baseline system WebPIE[7]. We ran both systems over the same dataset fpdLUBM8000. The results are shown in Table 4. Notice that the dataset is a fuzzy dataset, for WebPIE, we simply omitted the fuzzy degree, and submitted all crisp triples to the system. So our system (FuzzyPD) output a little more triples than WebPIE, because our system also updated 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 rational range. Thus we concluded that our system is comparable with the state-of-the-art inference system.

3) Scalability

To test the scalability of our algorithms, we ran 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 ran 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 6. 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 attributed this performance gain to the platform start-up 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 5 which contains the results of the second test. Figure 1 gives a direct illustration of the overhead effect. In Figure 1, if we subtracted 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.

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

TABLE 4 Experimental results of our system and WebPIE.

NUMBER OF UNIVERSITIES	TIME OF FUZZYPD (MINUTES)	TIME OF WEBPIE (MINUTES)
1000	38.8	41.32
2000	66.97	74.57
4000	110.40	130.87
8000	215.48	210.01

V. Related Work

Recently, there have been some works on learning fuzzy OWL ontologies [14], [15]. [5] is the first work to extend RDFS with fuzzy vagueness. In [6], we further proposed the fuzzy pD^* semantics which allows some useful OWL vocabularies, such as

TABLE 5 Scalability for fuzzy *pD reasoning.**

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

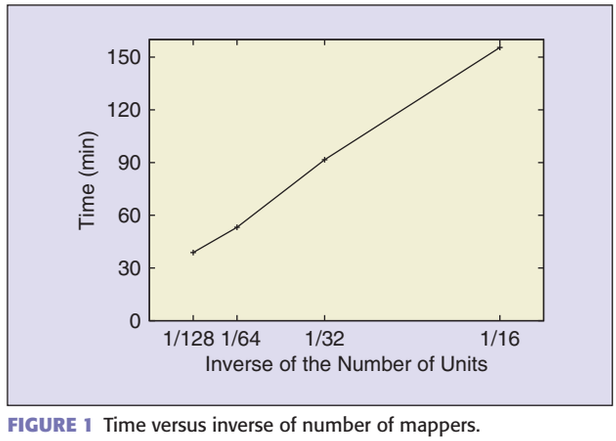


FIGURE 1 Time versus inverse of number of mappers.

TABLE 6 Scalability over data volume.

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

TransitiveProperty and SameAs. In [16] and [17], a more general framework for representing "annotated RDF data" and a query language called AnQL were proposed.

As far as we know, this is the first work that applies the MapReduce framework to tackle large scale reasoning in fuzzy OWL. The only work that tried to deal with large scale fuzzy ontologies was given in Pan et al. in [13]. Their work proposed a framework for fuzzy query answering in fuzzy DL-Lite. In contrast, our work focussed on the inference problem over large scale fuzzy pD^* ontologies.

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

Schlicht and Stuckenschmidt [18] showed peer-to-peer reasoning for the DL $\mathcal{ALC}$ but focusing on distribution rather than performance. Soma and Prasanna [19] presented a technique for parallel OWL inference through data partitioning. Experimental results showed good speedup but only on very small datasets (1M triples) and runtime performance was not reported.

In Weaver and Hendler [20], straightforward parallel RDFS reasoning on a cluster was presented. But this approach split the input to independent partitions. Thus it is only applicable for simple logics, e.g. RDFS without extending the RDFS schema, where the input is independent.

Newman et al. [21] decomposed and merged RDF molecules using MapReduce and Hadoop. They performed SPARQL queries on the data but performance was reported over a dataset of limited size (70,000 triples).

Urbani et al. [7] developed the MapReduce algorithms for materializing RDFS inference results. In [8], they further extended their methods to handle OWL pD^* fragment, and conducted experiment over a dataset containing 100 billion triples.

VI. 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 pD^* semantics with fuzzy vagueness). We first identified the major challenges to handle the fuzzy information when applying the MapReduce framework, and proposed a solution to tackle 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. They show the scalability of our main reasoning algorithm in both the dimensions of data volumes and number of processing units.

As a future work, we will apply our system to some applications, such as Genomics and multimedia data management. In another future work, we will consider other fuzzy semantics, such as the one based on type-2 fuzzy set theory [22], and propose MapReduce algorithms for the new semantics.

VII. Acknowledgments

Guilin Qi is partially supported by NSFC (61003157,60903010), Jiangsu Science Foundation (BK2010412), Excellent Youth Scholars Program of Southeast University under grant 4009001011, the Key Laboratory of Advanced Information Science and Network Technology of Beijing or the Key Laboratory of Information Science & Engineering of Railway Ministry(XDXX1011), the Key Laboratory of Computer Network and Information Integration (Southeast University).

References

[1] RDF. [Online]. Available: http://www.w3.org/RDF/

[2] RDFS. [Online]. Available: http://www.w3.org/TR/rdf-schema/

[3] H. J. Horst, "Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the owl vocabulary," *J. Web Semantics*, vol. 3, nos. 2–3, pp. 79–115, 2005.

[4] OWL. [Online]. Available: http://www.w3.org/TR/owl-features/

[5] U. Straccia, "A minimal deductive system for general fuzzy RDF," in *Proc. RR'09*, 2009, pp. 166–181.

[6] C. Liu, G. Qi, H. Wang, and Y. Yu, "Fuzzy reasoning over RDF data using OWL vocabulary," in *Proc. WI'11*, 2011.

[7] J. Urbani, S. Kotoulas, E. Oren, and F. Van Harmelen, "Scalable distributed reasoning using mapreduce," in *Proc. ISWC'09*, 2009, pp. 374–389.

[8] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal, "Owl reasoning with webpie: Calculating the closure of 100 billion triples," in *Proc. ESWC'10*, 2010, pp. 213–227.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI'04*, 2004, pp. 137–147.

[10] C. Liu, G. Qi, H. Wang, and Y. Yu, "Large scale fuzzy pd⋆ reasoning using mapreduce," in *Proc. ISWC'11*, 2011.

[11] DBPedia. [Online]. Available: http://dbpedia.org/

[12] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for owl knowledge base systems," *J. Web Semantics*, vol. 3, no. 2, pp. 158–182, 2005.

[13] J. Z. Pan, G. Stamou, G. Stoilos, S. Taylor, and E. Thomas, "Scalable querying services over fuzzy ontologies," in *Proc. WWW'08*, 2008, pp. 575–584.

[14] F. Zhang, Z. M. Ma, J. Cheng, and X. Meng, "Fuzzy semantic web ontology learning from fuzzy UML model," in *Proc. CIKM'09*, 2005.

[15] G. Da San Martino and A. Sperduti, "Mining structured data," *IEEE Comput. Intell. Mag.*, pp. 42–49, 2010.

[16] U. Straccia, N. Lopes, G. Lukacsy, and A. Polleres, "A general framework for representing and reasoning with annotated semantic web data," in *Proc. AAAI'10*. AAAI Press, 2010, pp. 1437–1442.

[17] U. S. N. Lopes, A. Polleres, and A. Zimmermann, "AnQL: SPARQLing up annotated RDF," in *Proc. ISWC'10*, 2010, pp. 518–533.

[18] A. Schlicht and H. Stuckenschmidt, "Peer-to-peer reasoning for interlinked ontologies," vol. 4, pp. 27–58, 2010.

[19] R. Soma and V. K. Prasanna, "Parallel inferencing for owl knowledge bases," in *Proc. ICPP'08*, 2008, pp. 75–82.

[20] J. Weaver and J. A. Hendler, "Parallel materialization of the finite RDFS closure for hundreds of millions of triples," in *Proc. ISWC'09*, 2009, pp. 682–697.

[21] A. Newman, Y.-F. Li, and J. Hunter, "Scalable semantics—The silver lining of cloud computing," in *Proc. ESCIENCE'08*, 2008.

[22] J. M. Mendel, "Type-2 fuzzy sets, a tribal parody," *IEEE Comput. Intell. Mag.*, pp. 24–27, 2010.