Toward Scalable Reasoning over Annotated RDF Data Using MapReduce

Chang Liu[1] Guilin Qi[2]

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

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.

Recently, there is an increasing interest to express additional information on top of RDF data. Several extensions of RDF were proposed in order to deal with time, uncertainty, trust and provenance. All these specific domains can be modeled by a general framework called *annotated RDF* data [3][5]. A recent work reported millions of triples with temporal information [1] and the number is still increasing. It is reasonable to expect more annotated RDF triples to be handled by semantic web applications. Therefore scalability will become an important issue for these applications.

Existing work [4] has shown that MapReduce is a scalable framework to perform large scale RDFS reasoning. This inspires us to solve the large scale annotated RDFS reasoning problem using MapReduce. We find that most of the optimizations for RDFS reasoning is also applicable for annotated RDFS reasoning. However, to reason with an arbitrary annotation domain, there are still some unique challenges that need to be handled. In this paper, we will discuss these challenges and solutions to tackle them. Our preliminary results show that our method is scalable for a specific domain: fuzzy domain.

2 Annotated RDFS Reasoning

2.1 Syntax

An annotated RDF triple is in form of $(s, p, o) : \lambda \in \mathbf{UBL} \times \mathbf{UB} \times \mathbf{UBL} \times D$[1]. Here $\mathbf{U}$, $\mathbf{B}$ and $\mathbf{L}$ are the sets of *URI references*, *blank nodes* and *literals* respectively. D is an *annotated domain* which is an idempotent commutative semi-ring $D = \langle L, \oplus, \otimes, \bot, \top \rangle$, where $\oplus$ is $\top$-annihilating, $\otimes$ is $\bot$-annihilating, and $\otimes$ is distributive over $\oplus$.

[1] Here we allow the predicate to be blank node to avoid the implicit typing rules.

2.2 Deductive System

Our discussion is based on the annotated extension of ρdf proposed in [5]. It is easy to extend to the full annotated RDFS semantics. For the annotated RDFS framework, there is a complete and sound rule set for an arbitrary annotation domain $D = \langle L, \oplus, \otimes, \bot, \top \rangle$. The rule set contains the following rules. Please note that the generalisation rule is destructive, *i.e.*, this rule removes the premises as the conclusion is inferred.

Table 1. Annotated RDFS Entailment Rules

Subproperty (a)	$(A, \texttt{sp}, B) : \lambda_1, (B, \texttt{sp}, C) : \lambda_2$	$(A, \texttt{sp}, C) : \lambda_1 \otimes \lambda_2$
Subproperty (b)	$(D, \texttt{sp}, E) : \lambda_1, (X, D, Y) : \lambda_2$	$(X, E, Y) : \lambda_1 \otimes \lambda_2$
Subclass (a)	$(A, \texttt{sc}, B) : \lambda_1, (B, \texttt{sc}, C) : \lambda_2$	$(A, \texttt{sc}, C) : \lambda_1 \otimes \lambda_2$
Subclass (b)	$(A, \texttt{sc}, B) : \lambda_1, (X, \texttt{type}, A) : \lambda_2$	$(X, \texttt{type}, B) : \lambda_1 \otimes \lambda_2$
Typing (a)	$(D, \texttt{dom}, B) : \lambda_1, (X, D, Y) : \lambda_2$	$(X, \texttt{type}, B) : \lambda_1 \otimes \lambda_2$
Typing (b)	$(D, \texttt{range}, B) : \lambda_1, (X, D, Y) : \lambda_2$	$(Y, \texttt{type}, B) : \lambda_1 \otimes \lambda_2$
Generalisation	$(X, A, Y) : \lambda_1, (X, A, Y) : \lambda_2$	$(X, A, Y) : \lambda_1 \oplus \lambda_2$

3 MapReduce Algorithm for Annotated RDFS Reasoning

3.1 Naive implementation

We will use typing rule (a) to illustrate how to use MapReduce program to encode a rule. To apply this rule, we essentially need to perform a join between $(D, \texttt{dom}, B) : \lambda_1$ and $(X, D, Y) : \lambda_2$. Algorithm 1 and Algorithm 2 provide the map and reduce function for this rule respectively. In this MapReduce program, the mappers scan all the annotated triples, and check for each triple if it has the form $(D, \texttt{dom}, B) : \lambda_1$ or $(X, D, Y) : \lambda_2$. If so, the mapper will emit a key/value pair where the key is D and the value is the triple. Then the reducers collect all triple pairs, and output the derived results.

Algorithm 1 map function for typing rule (a)

Input: key, triple
1: **if** triple.predicate == 'dom' **then**
2: emit({p=triple.subject}, {flag=0, u=triple.object, a=triple.annotation};
3: **end if**
4: emit({p=triple.predicate}, {flag=1, u=triple.subject, a=triple.annotation};

3.2 Challenges and solutions

Such a naive implementation, however, suffers severe efficiency problem. For example, the naive join implementation will introduce an expensive shuffling stage; arbitrary attempts of rule applications will result in fixpoint iterations. For this reason, previous work [4] for RDFS reasoning has proposed several optimizations. These optimizations include loading schema triples into memory, grouping data to avoid duplicates, and ordering the rule applications to avoid fixpoint iterations. For the annotated RDFS reasoning problem, most of these optimizations are also applicable. However, there are still additional challenges to deal with the annotation. In the following, we shall discuss these challenges and their solutions.

Algorithm 2 reduce function for typing rule (a)

Input: key, iterator values

```
1: set[0].clear(); set[1].clear();
2: for value ∈ values do
3:     set[value.flag].update(value.u, value.a);
4: end for
5: for (i, j) ∈ set[0]×set[1] do
6:     emit(null, new AnnotatedTriple(i.u, 'type', j.u, i.a⊗j.a));
7: end for
```

Generalization Rule One major difference between the general annotated RDFS reasoning and the RDFS reasoning is the generalization rule. In RDFS reasoning, we can use data grouping technique to avoid generating duplicates, and therefore the generalization rule is not necessary. In the general setting, however, this optimization is not applicable. We have to reproduce the whole dataset for this rule, which is expensive. On the other hand, such generalization rule is important to improve the efficiency in both time and space. Therefore the challenge is what is the best tradeoff. For annotated RDFS reasoning, the best decision is to apply the generalization rule twice: once at the beginning and once at the end. Furthermore, even though an explicit application of the generalization rule is unavoidable, the grouping technique is still useful to avoid duplicates as many as possible.

Unnecessary Derivation If applying a rule derives an annotated triple $\tau : \lambda$ where $\lambda = \bot$, such derivation is an *unnecessary derivation*. Unnecessary derivations will cause the reasoner to waste a lot of time to perform useless rule applications. Therefore we want to have as few unnecessary derivations as possible. To tackle this problem, we can design the map key according to the annotation such that two annotated triples, which definitely produce empty result, will not be grouped together. Therefore the reducers can generate fewer empty results. Notice that these optimizations for specific domains might not be applicable for the general setting. However, our general methods can handle the general annotated RDFS reasoning, even though they might be inefficient. Therefore we can treat these specific optimizations as 'plugins' to the general reasoner so that they do not affect the generality.

Fixpoint Calculation Calculating the complete results by applying the subclass rules and subproperty rules requires fixpoint iteration. For RDFS reasoning, as discussed in [4], we can load all schema triples into memory to solve this problem. The key problem is how to calculate the subclass (or subproperty) closure. In RDFS semantics, calculating the subclass (or subproperty) closure is essentially calculating the transitive closure over the subclass (or subproperty) graph. In the general setting, however, calculating the transitive closure is not enough: we have to deal with the annotations. To handle this challenge, we find that calculating the closure is essentially a variation of all-pairs shortest path calculation problem, *i.e.*, calculating the shortest pathes between each pair of

Table 2. 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

nodes in a weighted graph. We can modify the shortest path algorithm to deal with this problem.

4 Preliminary results

We have implemented a reasoner for fuzzy domain. In this section, we will report some preliminary results of our fuzzy RDFS reasoner. These results were originally reported in our journal paper [2].

We use Hadoop as our MapReduce platform. We randomly generate fuzzy degrees on top of the DBPedia core ontology as our dataset. After performing fuzzy RDFS reasoning algorithm, 133656 new fuzzy triples were derived from the original dataset which contains 26996983 fuzzy triples. Table 2 lists the running time results. The results show 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. Therefore we basically conclude that our method is scalable with respect to that the running time decreases as the number of units increases.

5 Conclusion

In this paper, we showed how to use MapReduce techniques to achieve scalable annotated RDFS reasoning. We discuss some major challenges that will cause efficiency issues, and propose several solutions to tackle them. We report our preliminary results over fuzzy RDFS dataset, and the results show that our method is scalable.

References

1. Hoffart, J., Suchanek, F.M., Berberich, K., Lewis-Kelham, E., de Melo, G., Weikum, G.: Yago2: exploring and querying world knowledge in time, space, context, and many languages. In: Proc. of WWW' 11. pp. 229–232 (2011)
2. Liu, C., Qi, G., Wang, H., Yu, Y.: Reasoning with Large Scale Ontologies in Fuzzy pD^* Using MapReduce. Computational Intelligence Magazine, IEEE 7(2), 54–66 (2012)
3. Udrea, O., Recupero, D., Subrahmanian, V.: Annotated rdf. ACM Transactions on Computational Logic 11(2), 1–41 (2010)
4. Urbani, J., Kotoulas, S., Oren, E., Harmelen, F.V.: Scalable distributed reasoning using mapreduce. In: Proc. of ISWC' 09. pp. 374 – 389 (2009)
5. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated Semantic Web data. Journal of Web Semantics 11, 72–95 (2012)