Scalable Nonmonotonic Reasoning over RDF data using MapReduce

Ilias Tachmazidis[1,2], Grigoris Antoniou[1,3], Giorgos Flouris[1], and Spyros Kotoulas[4]

[1] Institute of Computer Science, FORTH
[2] Department of Computer Science, University of Crete
[3] University of Huddersfield, UK
[4] IBM Research, IBM Ireland

Abstract. In this paper, we are presenting a scalable method for nonmonotonic rule-based reasoning over Semantic Web Data, using MapReduce. Our work is motivated by the recent unparalleled explosion of available data coming from the Web, sensor readings, databases, ontologies and more. Such datasets could benefit from the introduction of rule sets encoding commonly accepted rules or facts, application- or domain-specific rules, commonsense knowledge etc. This raises the question of whether, how, and to what extent knowledge representation methods are capable of handling huge amounts of data for these applications. We present a scalable MapReduce-based method for reasoning using defeasible stratified logics. Our results indicate that our method shows good scalability properties and is able to handle a benchmark dataset of 1 billion triples, bringing it on par with state-of-the-art methods for monotonic logics.

1 Introduction

Recently, we experience a significant growth of the amount of available data published on the Semantic Web. Billions of RDF triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, memory organizations like the British Museum and Europeanna, as well as news and entertainment sources such as BBC, are published, along with numerous vocabularies and conceptual schemas from e-science aiming to facilitate annotation and interlinking of scientific and scholarly data [24]. The recent rising of the Linked Open Data initiative[5] [6] is an answer to the need for such large and interconnected data. RDF(S) [18, 7] has become the de facto standard for representing such knowledge in the Semantic Web, due to its flexible and extensible representation of information, which is independent of the existence or absence of a schema, under the form of triples.

The amount, diversity and interlinkage of data published in this manner enables a new generation of decision making and business intelligence applications across domains. To fully exploit the immense value of such datasets and their interconnections, one should be able to reason over them using rule sets that allow the aggregation, visualization, understanding and exploitation of the raw data. Such reasoning is based on rules which capture the RDFS or OWL inference semantics, but also rules which encode

[5] `http://linkeddata.org/`

commonsense, domain-specific, or other practical knowledge that humans possess and would allow the system to automatically reach useful conclusions based on the provided data, i.e., infer new and useful knowledge based on the data and their interconnections.

The knowledge representation field has provided a rich set of semantics and techniques to use for reasoning using such rule sets, although the focus has been on complex knowledge structures and reasoning methods. On the other hand, RDF datasets are much simpler, but their size raises scalability challenges that cannot be addressed by standard approaches. For example, as described in [29], for 78,8 million statements crawled from the Web (a small percentage of the available knowledge), the number of inferred conclusions using the relatively simple RDFS ruleset consists of 1,5 billion triples; it is evident that coping with such amounts of data is impossible in standard, single-machine approaches due to both memory and performance issues.

To address this problem, the use of massive parallelism has been recently proposed [28, 22, 29, 14, 10, 30], where reasoning is handled by a set of machines, assigning each of them a part of the parallel computation. In some cases, this approach has allowed scaling reasoning up to 100 billion triples [28]. However, such approaches have focused on monotonic reasoning, or have not been evaluated in terms of scalability [19].

In this paper, we concentrate on nonmonotonic rule sets [2, 17]. Such rule sets provide additional benefits because they are more suitable for encoding commonsense knowledge and reasoning. In addition, in the case of poor quality data, monotonic logics such as RDFS cause an explosion of trivial (and often useless derivations), as also identified in [12]. The occurrence of low quality data is very common in the context of the Semantic Web [24], as data are fetched from different sources, which are not controlled by the data engineer; thus, nonmonotonic reasoning is more suitable for this context.

Our previous works [26, 27] described how defeasible logic reasoning [16], a commonly used nonmonotonic logic, can be implemented using massively parallel techniques. In [26, 27] we adopted the MapReduce framework [8], which is widely used for parallel processing of huge datasets. In particular, we used Hadoop, an open-source implementation of the MapReduce framework, with an extensive user list including companies like IBM, Yahoo!, Facebook and Twitter[6].

The approach of [27] addressed reasoning for stratified rule sets. Stratification is a well-known concept employed in many areas of knowledge representation for efficiency reasons, e.g., in tractable RDF query answering [25], Description Logics [4, 11, 20] and nonmonotonic formalisms [5], as it has been shown to reduce the computational complexity of various reasoning problems.

This paper is the first attempt evaluating the feasibility of applying nonmonotonic reasoning over RDF data using mass parallelization techniques. We present a technique for materialization using stratified defeasible logics, based on MapReduce and focussing on performance. A defeasible rule set for the LUBM[7] benchmark is presented, which is used to evaluate our approach. We present scalability results indicating that our approach scales superlinearly with the data size. In addition, since load-balancing is a significant performance inhibitor in reasoning systems [22], we show that our ap-

[6] http://wiki.apache.org/hadoop/PoweredBy

[7] http://swat.cse.lehigh.edu/projects/lubm/

Algorithm 1 Wordcount example

```
map(Long key, String value) :
   // key: position in document
   // value: document line
   for each word w in value
      EmitIntermediate(w, "1");

reduce(String key, Iterator values) :
   // key: a word
   // values : list of counts
   int count = 0;
   for each v in values
      count += ParseInt(v);
   Emit(key , count);
```

proach performs very well in this respect for the considered dataset, distributing data fairly uniformly across MapReduce tasks. Compared to our previous work with similar content, we extend [26] by considering multi-arity predicates, and improve [27] by experimenting over a standard RDF data benchmark (LUBM).

The rest of the paper is organized as follows. Section 2 introduces briefly the MapReduce Framework and Defeasible Logic. The algorithm for defeasible reasoning using MapReduce is described in Section 3, while Section 4 presents our experimental results. We conclude in Section 5.

2 Preliminaries

2.1 MapReduce

MapReduce is a framework for parallel processing over huge datasets [8]. Processing is carried out in two phases, a map and a reduce phase. For each phase, a set of user-defined map and reduce functions are run in a parallel fashion. The former performs a user-defined operation over an arbitrary part of the input and partitions the data, while the latter performs a user-defined operation on each partition.

MapReduce is designed to operate over key/value pairs. Specifically, each Map function receives a key/value pair and emits a set of key/value pairs. All key/value pairs produced during the map phase are grouped by their key and passed to the reduce phase. During the reduce phase, a $Reduce$ function is called for each unique key, processing the corresponding set of values.

Probably the most well-known MapReduce example is the *wordcount* example. In this example, we take as input a large number of documents and the final result is the calculation of the number of occurrences of each word. The pseudo-code for the Map and $Reduce$ functions is depicted in Algorithm 1.

During the map phase, each map operation gets as input a line of a document. The Map function extracts words from each line and emits that word *w* occurred once ("1").

Here we do not use the position of each line in the document, thus the *key* in Map is ignored. However, a word can be found more than once in a line. In this case we emit a <w, 1> pair for each occurrence. Consider the line "Hello world. Hello MapReduce.". Instead of emitting a pair <Hello, 2>, our simple example emits <Hello, 1> twice (pairs for words *world* and *MapReduce* are emitted as well). As mentioned above, the MapReduce framework will group and sort pairs by their key. Specifically for the word *Hello*, a pair <Hello, <1,1>> will be passed to the $Reduce$ function. The $Reduce$ function has to sum up all occurrence values for each word emitting a pair containing the word and the final number of occurrences. The final result for the word *Hello* will be <Hello, 2>.

2.2 Defeasible Logic - Syntax

A defeasible theory [21], [3] (a knowledge base in defeasible logic) consists of five different kinds of knowledge: facts, strict rules, defeasible rules, defeaters, and a superiority relation.

Facts are literals that are treated as known knowledge (given or observed facts).

Strict rules are rules in the classical sense: whenever the premises are indisputable (e.g., facts) then so is the conclusion. An example of a strict rule is "Emus are birds", which can be written formally as: "emu(X) $\rightarrow$ bird(X).".

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "Birds typically fly"; written formally: "bird(X) $\Rightarrow$ flies(X).".

Defeaters are rules that cannot be used to draw any conclusions. Their only use is to prevent some conclusions. An example is "If an animal is heavy then it might not be able to fly". Formally: "heavy(X) $\rightsquigarrow$ $\neg$flies(X).".

The *superiority relation* among rules is used to define priorities among rules, that is, where one rule may override the conclusion of another rule. For example, given the defeasible rules "r : bird(X) $\Rightarrow$ flies(X)" and "r$'$: brokenWing(X) $\Rightarrow$ $\neg$flies(X)" which contradict one another, no conclusive decision can be made about whether a bird with broken wings can fly. But if we introduce a superiority relation $>$ with r$'$ $>$ r, with the intended meaning that r$'$ is strictly stronger than r, then we can indeed conclude that the bird cannot fly.

Note that in this paper, the aforementioned term *literal* is defined strictly by the defeasible logic semantics. An RDF triple can be represented as a literal. However, considering the term *literal* as an RDF literal would be a common misunderstanding.

2.3 Defeasible Logic - Formal Definition

A rule r consists (a) of its antecedent (or body) A(r) which is a finite set of literals, (b) an arrow, and, (c) its consequent (or head) C(r) which is a literal. Given a set R of rules, we denote the set of all strict rules in R by R_s, and the set of strict and defeasible rules in R by R_{sd}. R[q] denotes the set of rules in R with consequent q. If q is a literal, $\sim$q denotes the complementary literal (if q is a positive literal p then $\sim$q is $\neg$p; and if q is $\neg$p, then $\sim$q is p)

A defeasible theory D is a triple (F,R,$>$) where F is a finite set of facts, R a finite set of rules, and $>$ a superiority relation upon R.

2.4 Defeasible Logic - Proof Theory

A conclusion of D is a tagged literal and can have one of the following four forms:

- $+\Delta$q, meaning that q is definitely provable in D.
- $-\Delta$q, meaning that q is not definitely provable in D (this does not necessarily mean that $\sim$q is definitely provable).
- $+\partial$q, meaning that q is defeasibly provable in D.
- $-\partial$q, meaning that q is not defeasibly provable in D (this does not necessarily mean that $\sim$q is defeasibly provable).

Provability is defined below. It is based on the concept of a derivation (or proof) in D = (F, R, $>$). A derivation is a finite sequence P = P(1), ..., P(n) of tagged literals satisfying the following conditions. The conditions are essentially inference rules phrased as conditions on proofs. P(1..$\imath$) denotes the initial part of the sequence P of length i. For more details on provability and an explanation of the intuition behind the conditions below, see [16].

$+\Delta$: We may append P($\imath$ + 1) = $+\Delta$q if either
q $\in$ F or
$\exists$r $\in$ R_s[q] $\forall\alpha \in$ A(r): $+\Delta\alpha \in$ P(1..$\imath$)

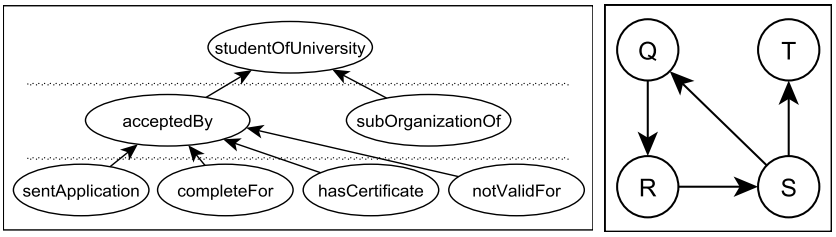
$-\Delta$: We may append P($\imath$ + 1) = $-\Delta$q if
q $\notin$ F and
$\forall$r $\in$ R_s[q] $\exists\alpha \in$ A(r): $-\Delta\alpha \in$ P(1..$\imath$)

$+\partial$: We may append P ($\imath$ + 1) = $+\partial$q if either
(1) $+\Delta$q $\in$ P(1..$\imath$) or
(2) (2.1) $\exists$r $\in$ R_{sd}[q] $\forall\alpha \in$ A(r): $+\partial\alpha \in$ P(1..$\imath$) and
(2.2) $-\Delta \sim$q $\in$ P(1..$\imath$) and
(2.3) $\forall$s $\in$ R[$\sim$q] either
(2.3.1) $\exists\alpha \in$ A(s): $-\partial\alpha \in$ P(1..$\imath$) or
(2.3.2) $\exists$t $\in$ R_{sd}[q] such that
$\forall\alpha \in$ A(t): $+\partial\alpha \in$ P(1..$\imath$) and t $>$ s

$-\partial$: We may append P($\imath$ + 1) = $-\partial$q if
(1) $-\Delta$q $\in$ P(1..$\imath$) and
(2) (2.1) $\forall$r $\in$ R_{sd}[q] $\exists\alpha \in$ A(r): $-\partial\alpha \in$ P(1..$\imath$) or
(2.2) $+\Delta \sim$q $\in$ P(1..$\imath$) or
(2.3) $\exists$s $\in$ R[$\sim$q] such that
(2.3.1) $\forall\alpha \in$ A(s): $+\partial\alpha \in$ P(1..$\imath$) and
(2.3.2) $\forall$t $\in$ R_{sd}[q] either
$\exists\alpha \in$ A(t): $-\partial\alpha \in$ P(1..$\imath$) or t $\not>$ s

3 Algorithm description

The algorithm that is described in this section, shows how parallel reasoning can be performed using the MapReduce framework. Parallel reasoning can be based either on



(a) Stratified (b) Non-stratified

Fig. 1: Predicate dependency graph

rule partitioning or on data partitioning [13]. Rule partitioning assigns the computation of each rule to a computer in the cluster. However, balanced work distribution in this case is difficult to achieve, as the computational burden per rule (and node) depends on the structure of the rule set. On the other hand, data partitioning assigns a subset of data to each computer. Data partitioning is more flexible, providing more fine-grained partitioning and allowing easier distribution among nodes in a balanced manner. Our approach is based on data partitioning.

For reasons that will be explained later, defeasible reasoning over rule sets with multi-argument predicates is based on the dependencies between predicates which is encoded using the *predicate dependency graph*. Thus, rule sets can be divided into two categories: *stratified* and *non-stratified*. Intuitively, a *stratified* rule set can be represented as an acyclic hierarchy of dependencies between predicates, while a *non-stratified* cannot. We address the problem for stratified rule sets by providing a well-defined reasoning sequence, and explain at the end of the section the challenges for non-stratified rule sets.

The dependencies between predicates can be represented using a *predicate dependency graph*. For a given rule set, the *predicate dependency graph* is a directed graph whose:

- vertices correspond to predicates. For each literal p, both p and $\neg p$ are represented by the positive predicate.
- edges are directed from a predicate that belongs to the body of a rule, to a predicate that belongs to the head of the same rule. Edges are used for all three rule types (strict rules, defeasible rules, defeaters).

Stratified rule sets (correspondingly, *non-stratified rule sets*) are rule sets whose predicate dependency graph is acyclic (correspondingly, contains a cycle). *Stratified theories* are theories based on stratified rule sets. Figure 1a depicts the predicate dependency graph of a stratified rule set, while Figure 1b depicts the predicate dependency graph of a non-stratified rule set. The superiority relation is not part of the graph.

As an example of a stratified rule set, consider the following:

r1: X *sentApplication* A, A *completeFor* D $\Rightarrow$ X *acceptedBy* D.
r2: X *hasCertificate* C, C *notValidFor* D $\Rightarrow$ X $\neg$*acceptedBy* D.
r3: X *acceptedBy* D, D *subOrganizationOf* U $\Rightarrow$ X *studentOfUniversity* U.
r1 $>$ r2.

The predicate dependency graph for the above rule set is depicted in Figure 1a. The predicate graph can be used to determine strata for the different predicates. In particular, predicates (nodes) with no outgoing edges are assigned the maximum stratum, which is equal to the maximum depth of the directed acyclic graph (i.e., the size of the maximum path that can be defined through its edges), say k. Then, all predicates that are connected with a predicate of stratum k are assigned stratum $k-1$, and the process continues recursively until all predicates have been assigned some stratum. Note that predicates are reassigned to a lower stratum in case of multiple dependencies. The dashed horizontal lines in Figure 1a are used to separate the various strata, which, in our example, are as follows:

Stratum 2: *studentOfUniversity*
Stratum 1: *acceptedBy*, *subOrganizationOf*
Stratum 0: *sentApplication*, *completeFor*, *hasCertificate*, *notValidFor*

Stratified theories are often called decisive in the literature [5].

Proposition 1. *[5] If D is stratified, then for each literal p:*
(a) either $D \vdash +\Delta p$ *or* $D \vdash -\Delta p$
(b) either $D \vdash +\partial p$ *or* $D \vdash -\partial p$

Thus, there are three possible states for each literal p in a stratified theory: (a) $+\Delta p$ *and* $+\partial p$, (b) $-\Delta p$ *and* $+\partial p$ and (c) $-\Delta p$ *and* $-\partial p$.

Reasoning is based on facts. According to defeasible logic algorithm, facts are $+\Delta$ and every literal that is $+\Delta$, is $+\partial$ too. Having $+\Delta$ and $+\partial$ in our initial knowledge base, it is convenient to store and perform reasoning only for $+\Delta$ and $+\partial$ predicates.

This representation of knowledge allows us to reason and store provability information regarding various facts more efficiently. In particular, if a literal is not found as a $+\Delta$ (correspondingly, $+\partial$) then it is $-\Delta$ (correspondingly, $-\partial$). In addition, stratified defeasible theories have the property that if we have computed all the $+\Delta$ and $+\partial$ conclusions up to a certain stratum, and a rule whose body contains facts of said stratum does not currently fire, then this rule will also be inapplicable in subsequent passes; this provides a well-defined reasoning sequence, namely considering rules from lower to higher strata.

3.1 Reasoning overview

During reasoning we will use the representation (<fact, ($+\Delta$,$+\partial$)>) to store our inferred facts. We begin by transforming the given facts, in a single MapReduce pass, into (<fact, ($+\Delta$,$+\partial$)>).

Now let us consider for example the facts "John *sentApplication* App", "App *completeFor* Dep", "John *hasCertificate* Cert", "Cert *notValidFor* Dep" and "Dep *subOrganizationOf* Univ" . The *initial pass* on these facts using the aforementioned rule set will create the following output:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>

No reasoning needs to be performed for the lowest stratum (stratum 0) since these predicates (*sentApplication*, *completeFor*, *hasCertificate*, *notValidFor*) do not belong to the head of any rule. As is obvious by the definition of $+\partial$, $-\partial$, defeasible logic introduces uncertainty regarding inference, because certain facts/rules may "block" the firing of other rules. This can be prevented if we reason for each stratum separately, starting from the lowest stratum and continuing to higher strata. This is the reason why for a hierarchy of N strata we have to perform $N - 1$ times the procedure described below. In order to perform defeasible reasoning we have to run two passes for each stratum. The first pass computes which rules can fire. The second pass performs the actual reasoning and computes for each literal if it is definitely or defeasibly provable. The reasons for both decisions (reasoning sequence and two passes per stratum) are explained in the end of the next subsection.

3.2 Pass #1: Fired rules calculation

During the first pass, we calculate the inference of fired rules, which is performed by joining predicates on common argument values. Such techniques for basic and multi-way join have been described in [9] and optimized in [1]. In order to achieve an efficient implementation, optimizations in [1] should be taken into consideration. Here we elaborate on our approach for basic joins and explain at the end of the subsection how it can be generalized for multi-way joins.

Basic join is performed on common argument values. Consider the following rule:

r1: X *sentApplication* A, A *completeFor* D $\Rightarrow$ X *acceptedBy* D.

The key observation is that "X *sentApplication* A" and "A *completeFor* D" can be joined on their common argument *A*. Based on this observation, during the Map operation, we emit pairs of the form <A, (X, *sentApplication*)> for predicate *sentApplication* and <A, (D, *completeFor*)> for predicate *completeFor*. The idea is to join *sentApplication* and *completeFor* only for literals that have the same value on argument *A*. During the $Reduce$ operation we combine *sentApplication* and *completeFor* producing *acceptedBy*.

In our example, the facts "John *sentApplication* App" and "App *completeFor* Dep" will cause Map to emit <App, (John, *sentApplication*)> and <App, (Dep, *completeFor*)>. The MapReduce framework groups and sorts intermediate pairs passing <App, <(John, *sentApplication*), (Dep, *completeFor*)>> to the $Reduce$ operation. Finally, at $Reduce$ we combine given values and infer "John *acceptedBy* Dep".

To support defeasible logic rules which have blocking rules, this approach must be extended. We must record all fired rules prior to any conclusion inference, whereas for monotonic logics this is not necessary, and conclusion derivation can be performed immediately. The reason why this is so is explained at the end of the subsection. Pseudo-code for Map and $Reduce$ functions, for a basic join, is depicted in Algorithm 2. Map function reads input of the form <literal, ($+\Delta$, $+\partial$)> or <literal, ($+\partial$)> and emits pairs of the form <matchingArgumentValue, (nonMatchingArgumentValue, *Predicate*, $+\Delta$, $+\partial$)> or <matchingArgumentValue, (nonMatchingArgumentValue, *Predicate*, $+\partial$)> respectively.

Algorithm 2 Fired rules calculation

```
map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (derived conclusion)
  For every common argumentValue in value
    EmitIntermediate(argumentValue, value);

reduce(String key, Iterator values):
  // key: matching argument
  // value: literals for matching
  For every argument value match in values
    If strict rule fired with all premises being +Δ then
      Emit(firedLiteral, "[¬,] +Δ, +∂, ruleID");
    else
      Emit(firedLiteral, "[¬,] +∂, ruleID");
```

Now consider again the stratified rule set described in the beginning of the section, for which the *initial pass* will produce the following output:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>

We perform reasoning for stratum 1, so we will use as premises all the available information for predicates of stratum 0. The Map function will emit the following pairs:

<App, (John,*sentApplication*,$+\Delta$,$+\partial$)> <App, (Dep,*completeFor*,$+\Delta$,$+\partial$)>
<Cert, (John,*hasCertificate*,$+\Delta$,$+\partial$)> <Cert, (Dep,*notValidFor*,$+\Delta$,$+\partial$)>

The MapReduce framework will perform grouping/sorting resulting in the following intermediate pairs:

<App, <(John,*sentApplication*,$+\Delta$,$+\partial$), (Dep,*completeFor*,$+\Delta$,$+\partial$)>>
<Cert, <(John,*hasCertificate*,$+\Delta$,$+\partial$), (Dep,*notValidFor*,$+\Delta$,$+\partial$)>>

During reduce we combine premises in order to emit the *firedLiteral* which consists of the fired rule head predicate and the *nonMatchingArgumentValue* of the premises. However, inference depends on the type of the rule. In general, for all three rule types (strict rules, defeasible rules and defeaters) if a rule fires then we emit as output <firedLiteral, ([¬,] $+\partial$, ruleID)> ([¬,] denotes that "¬" is optional and appended only if the *firedLiteral* is negative). However, there is a special case for strict rules. This special case covers the required information for $+\Delta$ conclusions inference. If all premises are $+\Delta$ then we emit as output <firedLiteral, ([¬,]$+\Delta$,$+\partial$,ruleID)> instead of <firedLiteral, ([¬,]$+\partial$, ruleID)>.

For example, during the reduce phase the reducer with key:

App will emit <John *acceptedBy* Dep, ($+\partial$, r1)>
Cert will emit <John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

As we see here, "John *acceptedBy* Dep" and "John $\neg$*acceptedBy* Dep" are computed by different reducers (with key *App* and *Cert* respectively) which do not communicate with each other. Thus, none of the two reducers has all the available information in order to perform defeasible reasoning. Therefore, we need a second pass for the reasoning.

Let us illustrate why reasoning has to be performed for each stratum separately, requiring stratified rule sets. Consider again our running example. We will attempt to perform reasoning for all the strata simultaneously. On the one hand, we cannot join "John *acceptedBy* Dep" with "Dep *subOrganizationOf* Univ" prior to the second pass because we do not have a final conclusion on "John *acceptedBy* Dep". Thus, we will not perform reasoning for "John *studentOfUniversity* Univ" during the second pass, which leads to data loss. On the other hand, if another rule (say r4) supporting "John $\neg$*studentOfUniversity* Univ" had also fired, then during the second pass, we would have mistakenly inferred "John $\neg$*studentOfUniversity* Univ", leading our knowledge base to inconsistency.

In case of multi-way joins we compute the head of the rule (*firedLiteral*) by performing joins, on common argument values, in one or more MapReduce passes as explained in [9] and [1]. As above, for each fired rule, we must take into consideration the type of the rule and whether all the premises are $+\Delta$ or not. Finally, the format of the output remains the same (<firedLiteral, ([$\neg$,] $+\Delta$, $+\partial$, ruleID)> or <firedLiteral, ([$\neg$,] $+\partial$, ruleID)>).

3.3 Pass #2: Defeasible reasoning

We proceed with the second pass. Once fired rules are calculated, a second MapReduce pass performs reasoning for each literal separately. We should take into consideration that each literal being processed could already exist in our knowledge base (due to the *initial pass*). In this case, we perform duplicate elimination by not emitting pairs for existing conclusions. The pseudo-code for Map and $Reduce$ functions, for stratified rule sets, is depicted in Algorithm 3.

After both *initial pass* and fired rules calculation (first pass), our knowledge base will consist of:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)> <John *acceptedBy* Dep, ($+\partial$, r1)>
<John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

During the Map operation we must first extract from $value$ the literal and the inferred knowledge or the fired rule using *extractLiteral()* and *extractKnowledge()* respectively. For each literal p, both p and $\neg p$ are sent to the same reducer. The "$\neg$" in *knowledge* distinguishes p from $\neg p$. The Map function will emit the following pairs:

<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)> <John *acceptedBy* Dep, ($+\partial$, r1)>
<John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

Algorithm 3 Defeasible reasoning

```
map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: inferred knowledge/fired rules
  String p = extractLiteral(value);
  If p does not belong to current stratum then
    return;
  String knowledge = extractKnowledge(value);
  EmitIntermediate(p, knowledge);

reduce(String p, Iterator values) :
  // p: a literal
  // values : inferred knowledge/fired rules
  For each value in values
    markKnowledge(value);
  For literal in {p, ¬p} check
    If literal is already +Δ then
      return;
    Else If strict rule fired with all premises being +Δ then
      Emit(literal, "+Δ, +∂");
    Else If literal is +∂ after defeasible reasoning then
      Emit(literal, "+∂");
```

MapReduce framework will perform grouping/sorting resulting in the following intermediate pairs:

<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>
<John *acceptedBy* Dep, <($+\partial$, r1), ($\neg$,$+\partial$, r2)>>

For the Reduce, the key contains the literal and the values contain all the available information for that literal (known knowledge, fired rules). We traverse over *values* marking known knowledge and fired rules using the *markKnowledge()* function. Subsequently, we use this information in order to perform reasoning for each literal.

During the reduce phase the reducer with key:

"Dep *subOrganizationOf* Univ" will not emit anything
"John *acceptedBy* Dep" will emit <John *acceptedBy* Dep, ($+\partial$)>

The literal "Dep *subOrganizationOf* Univ" is known knowledge. For known knowledge a potential duplicate elimination must be performed. We reason simultaneously both for "John *acceptedBy* Dep" and "John $\neg$*acceptedBy* Dep". As "John $\neg$*acceptedBy* Dep" is $-\partial$, it does not need to be recorded.

3.4 Final remarks

The total number of MapReduce passes is independent of the size of the given data, and is determined by the form of the rules, in particular by the number of strata that the

rules are stratified into. As mentioned in subsection 3.2, performing reasoning for each stratum separately eliminates data loss and inconsistency, thus our approach is sound and complete since we fully comply with the defeasible logic provability. Eventually, our knowledge base consists of $+\Delta$ and $+\partial$ literals.

The situation for non-stratified rule sets is more complex. Reasoning can be based on the algorithm described in [15], performing reasoning until no new conclusion is derived. However, the total number of required passes is generally unpredictable, depending both on the given rule set and the data distribution. Additionally, an efficient mechanism for "$\forall$r $\in$ R_s[q] $\exists\alpha \in$ A(r): $-\Delta\alpha \in$ P(1..i)" (in $-\Delta$ provability) and "$\forall$r $\in$ R_{sd}[q] $\exists\alpha \in$ A(r): $-\partial\alpha \in$ P(1..i)" (in 2.1 of $-\partial$ provability) computation is yet to be defined because all the available information for the literal must be processed by a single node (since nodes do not communicate with each other), causing either main memory insufficiency or skewed load balancing decreasing the parallelization. Finally, we have to reason for and store every possible conclusion ($+\Delta$,$-\Delta$,$+\partial$,$-\partial$), producing a significantly larger stored knowledge base.

4 Evaluation

In this Section, we are presenting the methodology, dataset and experimental results for an implementation of our approach using Hadoop.

Methodology. Our evaluation is centered around scalability and the capacity of our system to handle large datasets. In line with standard practice in the field of high-performance systems, we have defined scalability as the ability to process datasets of increasing size in a proportional amount of time and the ability of our system to perform well as the computational resources increase. With regard to the former, we have performed experiments using datasets of various sizes (yet similar characteristics).

With regard to scaling computational resources, it has been empirically observed that the main inhibitor of parallel reasoning systems has been load-balancing between compute nodes [14]. Thus, we have also focused our scalability evaluation on this aspect.

The communication model of Hadoop is not sensitive to the physical location of each data partition. In our experiments, Map tasks only use local data (implying very low communication costs) and Reduce operates using hash-partitioning to distribute data across the cluster (resulting in very high communication costs regardless of the distribution of data and cluster size). In this light, scalability problems do not arise by the number of compute nodes, but by the unequal distribution of the workload in each reduce task. As the number of compute nodes increases, this unequal distribution becomes visible and hampers performance.

Dataset. We have used the most popular benchmark for reasoning systems, LUBM. LUBM allows us to scale the size of the data to an arbitrary size while keeping the reasoning complexity constant. For our experiments, we generated up to 8000 universities resulting in approximately 1 billion triples.

Rule set. The logic of LUBM can be partially expressed using RDFS and OWL2-RL. Nevertheless, neither of these logics are defeasible. Thus, to evaluate our system, we have created the following ruleset:

r1: X rdf:type FullProfessor → X rdf:type Professor.
r2: X rdf:type AssociateProfessor → X rdf:type Professor.
r3: X rdf:type AssistantProfessor → X rdf:type Professor.
r4: P publicationAuthor X, P publicationAuthor Y → X commonPublication Y.
r5: X teacherOf C, Y takesCourse C → X teaches Y.
r6: X teachingAssistantOf C, Y takesCourse C → X teaches Y.
r7: X commonPublication Y → X commonResearchInterests Y.
r8: X hasAdvisor Z, Y hasAdvisor Z → X commonResearchInterests Y.
r9: X hasResearchInterest Z, Y hasResearchInterest Z → X commonResearchInterests Y.
r10: X hasAdvisor Y ⇒ X canRequestRecommendationLetter Y.
r11: Y teaches X ⇒ X canRequestRecommendationLetter Y.
r12: Y teaches X, Y rdf:type PostgraduateStudent ⇒ X ¬canRequestRecommendationLetter Y.
r13: X rdf:type Professor, X worksFor D, D subOrganizationOf U ⇒ X canBecomeDean U.
r14: X rdf:type Professor, X headOf D, D subOrganizationOf U ⇒ X ¬canBecomeDean U.
r15: X worksFor D ⇒ X canBecomeHeadOf D.
r16: X worksFor D, Z headOf D, X commonResearchInterests Z ⇒ X ¬canBecomeHeadOf D.
r17: Y teaches X ⇒ X suggestAdvisor Y.
r18: Y teaches X, X hasAdvisor Z ⇝ X ¬suggestAdvisor Y.
r12 > r11, r14 > r13, r16 > r15, r18 > r17.

MapReduce jobs description. We need 8 jobs in order to perform reasoning on the above rule set. The *first* job is the *initial pass* described in Section 3 (which we also use to compute rules r1-r3). For the rest of the jobs, we first compute fired rules and then perform reasoning for each stratum separately. The *second* job computes rules r4-r6. During the *third* job we perform duplicate elimination, since r4-r6 are strict rules. We compute rules r7-r14 during the *fourth* job while reasoning on them, is performed during the *fifth* job. Jobs *six* and *seven* compute rules r15-r18. Finally, during the *eighth* job we perform reasoning on r15-r18, finishing the whole procedure.

Platform. Our experiments were performed on a IBM x3850 server with 40 cores and 750GB of RAM, connected to a XIV Storage Area Network (SAN), using a 10Gbps storage switch. We have used IBM Hadoop Cluster v1.3, which is compatible with Hadoop v0.20.2, along with an optimization to reduce Map task overhead, in line with [23]. Although our experiments were run on a single machine, there was no direct communication between processes and all data was transferred through persistent storage. We have used a number of Mappers and Reducers equal to the number of cores in the system (i.e. 40).

Results. Figure 2 shows the runtimes of our system for varying input sizes. We make the following observations: (a) even for a single node, our system is able to handle very large datasets, easily scaling to 1 billion triples. (b) The scaling properties with regard to dataset size are excellent: in fact, as the size of the input increases, the throughput of our system increases. For example, while our system can process a dataset of 125 million triples at a throughput of 27Ktps, for 1 billion triples, the throughput becomes 63Ktps. This is attributed to the fact that job startup costs are amortized over the longer runtime of the bigger datasets.

The above show that our system is indeed capable of achieving high performance and scales very well with the size of the input. Nevertheless, to further investigate how

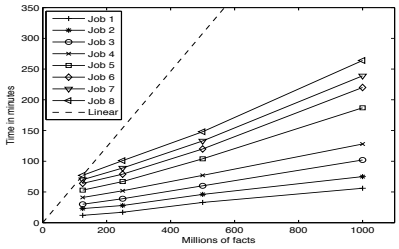
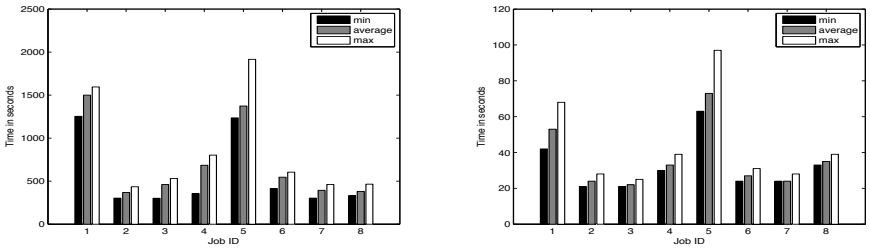


Fig. 2: Runtime in minutes for various datasets, and projected linear scalability. Job runtimes are stacked (i.e. runtime for Job 8 includes the runtimes for Jobs 1-7).



(a) 40 reduce tasks

(b) 400 reduce tasks

Fig. 3: Minimum, average and maximum reduce task runtime for each job.

our system would perform when the data size precludes the use of a single machine, it is critical to examine the load-balancing properties of our algorithm.

As previously described, in typical MapReduce applications, load-balancing problems arise during the reduce phase. Namely, it is possible that the partitions of the data processed in a single reduce task vary widely in terms of compute time required. This is a potential scalability bottleneck. To test our system for such issues, we have launched an experiment where we have increased the number of reduce tasks to 400. We can expect that, if the load balance for 400 reduce tasks is relatively uniform, our system is able to scale at least to that size.

Figure 3 shows the load balance between different reduce tasks, for 1 billion triples and 40 (Figure 3a) or 400 (Figure 3b) reduce tasks. In principle, an application performs badly when a single task dominates the runtime, since all other tasks would need to wait for it to finish. In our experiments, it is evident that no such task exists. In addition, one may note that the system is actually faster with 400 reduce tasks. This is attributed both to the fact that each core in our platform can process two threads in parallel, and to implementation aspects of Hadoop that result in tasks, processing approximately 1GB, demonstrating higher throughput than larger tasks.

Although a direct comparison is not meaningful, the throughput of our system is in line with results obtained when doing monotonic reasoning using state of the art RDF stores and inference engine. For example, OWLIM claims a 14.4-hour loading time

for the same dataset when doing OWL horst inference [8]. WebPIE [28], which is also based on MapReduce, presents an OWL-horst inference time of 35 minutes, albeit on 64 lower-spec nodes and requiring an additional dictionary encoding step.

Given the significant overhead of nonmonotonic reasoning, and in particular, the fact that inferences can not be drawn directly, this result is counter-intuitive. The key to the favorable performance of our approach is that the "depth" of the reasoning is fixed, on a per rule set basis. The immediate consequence is that the number of MapReduce jobs, which bear significant startup costs, is also fixed. In other words, the "predictable" nature of stratified logics allows us to have less complicated relationships between facts in the system.

Finally, we should take into consideration the fact that LUBM produces fairly uniform data. Although there is significant skew in LUBM (e.g. in the frequency of terms such as rdf:type), the rule set that we have used in the evaluation does not perform joins on such highly skewed terms. However, our previous work [27] shows that our approach can cope with highly skewed data, which follow a zipf distribution.

5 Conclusion and Future Work

In this paper we studied the feasibility of nonmonotonic rule systems over large volumes of semantic data. In particular, we considered defeasible reasoning over RDF, and ran experiments over RDF data. Our results demonstrate that such reasoning scales very well. In particular, we have shown that nonmonotonic reasoning is not only possible, but can compete with state-of-the-art monotonic logics. To the best of our knowledge, this is the first study demonstrating the feasibility of inconsistency-tolerant reasoning over RDF data using mass parallelization techniques.

In future work, we intend to perform an extensive experimental evaluation in order to verify our results for different input dataset morphologies. In addition, we plan to apply the MapReduce framework to ontology dynamics (including evolution, diagnosis, and repair) approaches based on validity rules (integrity constraints). These problems are closely related to inconsistency-tolerant reasoning, as violation of constraints may be viewed as a logical inconsistency.

6 Acknowledgments

This work was partially supported by the PlanetData NoE (FP7:ICT-2009.3.4, #257641).

References

1. Afrati, F.N., Ullman, J.D.: Optimizing joins in a mapreduce environment. In: EDBT (2010)
2. Antoniou, G., van Harmelen, F.: A Semantic Web Primer, 2nd Edition. The MIT Press, 2 edn. (March 2008)
3. Antoniou, G., Williams, M.A.: Nonmonotonic reasoning. MIT Press (1997)

[8] http://www.ontotext.com/owlim/benchmark-results/lubm

4. Baader, F., Kosters, R.: Nonstandard Inferences in Description Logics: The Story So Far. Mathematical Problems from Applied Logic I, volume 4 of International Mathematical Series (2006)
5. Billington, D.: Defeasible Logic is Stable. J. Log. Comput. 3(4), 379–400 (1993)
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)
7. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. www.w3.org/TR/2004/REC-rdf-schema-20040210 (2004)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters
9. Fische, F.: Investigation & Design for Rule-based Reasoning. Tech. rep., LarKC (2010)
10. Goodman, E.L., Jimenez, E., Mizell, D., Al-Saffar, S., Adolf, B., Haglin, D.J.: High-Performance Computing Applied to Semantic Databases. In: ESWC (2). pp. 31–45 (2011)
11. Haase, C., Lutz, C.: Complexity of Subsumption in the EL Family of Description Logics: Acyclic and Cyclic TBoxes. In: ECAI-08. pp. 25–29 (2008)
12. Hogan, A., Harth, A., Polleres, A.: Scalable Authoritative OWL Reasoning for the Web. Int. J. Semantic Web Inf. Syst. 5(2), 49–90 (2009)
13. Kotoulas, S., van Harmelen, F., Weaver, J.: KR and Reasoning on the Semantic Web: Web-Scale Reasoning (2011)
14. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: WWW. pp. 531–540 (2010)
15. Maher, M.J., Rock, A., Antoniou, G., Billington, D., Miller, T.: Efficient Defeasible Reasoning Systems. IJAIT 10, 2001 (2001)
16. Maher, M.J.: Propositional Defeasible Logic has Linear Complexity. CoRR cs.AI/0405090 (2004)
17. Maluszynski, J., Szalas, A.: Living with Inconsistency and Taming Nonmonotonicity. In: Datalog. pp. 384–398 (2010)
18. Manola, F., Miller, E., McBride, B.: RDF Primer. www.w3.org/TR/rdf-primer (2004)
19. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Description Logics (2010)
20. Nebel, B.: Terminological Reasoning is Inherently Intractable. Artificial Intelligence 43, 235–249 (1990)
21. Nute, D.: Defeasible Logic. In: Handbook of Logic in Artificial Intelligence and Logic Programming-Nonmonotonic Reasoning and Uncertain Reasoning(Volume 3) (1994)
22. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. J. Web Sem. 7(4), 305–316 (2009)
23. R. Vernica, A. Balmin, K.B., Ercegovac, V.: Adaptive Mapreduce using Situation-Aware Mappers. In: EDBT
24. Roussakis, Y., Flouris, G., Christophides, V.: Declarative Repairing Policies for Curated KBs. In: HDMS (2011)
25. Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and Minimization of RDF(S) Query Patterns. In: ISWC-05 (2005)
26. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S.: Towards Parallel Nonmonotonic Reasoning with Billions of Facts. In: KR-12 (2012)
27. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S., McCluskey, L.: Large-scale Parallel Stratified Defeasible Reasoning. In: ECAI-12 (2012)
28. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: OWL reasoning with webPIE: Calculating the Closure of 100 Billion Triples. In: ESWC (1). pp. 213–227 (2010)
29. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: ISWC. pp. 634–649 (2009)
30. Weaver, J., Hendler, J.A.: Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In: International Semantic Web Conference. pp. 682–697 (2009)