Large-scale Parallel Stratified Defeasible Reasoning

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

Abstract. We are recently experiencing an unprecedented explosion of available data coming from the Web, sensors readings, scientific databases, government authorities 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. In this paper, we consider inconsistency-tolerant reasoning in the form of defeasible logic, and analyze how parallelization, using the MapReduce framework, can be used to reason with defeasible rules over huge datasets. We extend previous work by dealing with predicates of arbitrary arity, under the assumption of stratification. Moving from unary to multi-arity predicates is a decisive step towards practical applications, e.g. reasoning with linked open (RDF) data. Our experimental results demonstrate that defeasible reasoning with millions of data is performant, and has the potential to scale to billions of facts.

1 Introduction

Currently, we experience a significant growth of the amount of available data originating from sensor readings, scientific databases, government authorities etc. Such data are mainly published on the Web, providing easier knowledge exchange and interlinkage [16]. This yields the need for large and interconnected data, as shown by the Linked Open Data initiative [4].

The study of knowledge representation has been mainly targeted on complex knowledge structures and reasoning methods for processing such structures. This raises the question whether such reasoning methods can be applied on huge datasets. Reasoning should be performed using rule sets that would allow the aggregation, visualization, understanding and exploitation of given datasets and their interconnections. Specifically, one should use rules able to encode inference semantics, as well as commonsense and practical conclusions in order to infer new and useful knowledge based on the data. This is usually a formidable task when it comes to web-scale data: for example, as described in [19] for 78,8 million statements crawled from the Web, the number of inferred conclusions (RDFS closure) consists of 1,5 billion triples.

In this work, we study nonmonotonic rule sets [2], [13] which are suitable for encoding commonsense knowledge and reasoning. In addition, nonmonotonic rule sets provide supplementary advantages in the case of poor quality data, as they can prevent triviality of inference. The occurrence of low quality data is common when they are fetched from different sources, which are not controlled by the data engineer.

Over the last years, parallel reasoning has been studied extensively e.g., in [15], [19], [9], [8], scaling reasoning up to 100 billion triples [18]. These works address the problem by using parallel reasoning techniques that allow simultaneous processing over distinct chunks of data, with each chunk being assigned to a computer in the cloud.

Parallel reasoning can be based either on rule partitioning or on data partitioning [10]. Rule partitioning assigns the computation of each rule to a computer in the cloud. 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 in the cloud. Data partitioning is more flexible, providing more fine-grained partitioning and allowing easier distribution among nodes in a balanced manner.

Current parallelization approaches have focused on monotonic reasoning, such as RDFS and OWL-horst, or have not been evaluated in terms of scalability [14]. Our paper deals with nonmonotonic rules and reasoning, and is therefore novel. Nonmonotonic reasoning has been chosen because it allows to overcome triviality of reasoning caused by inconsistent or incomplete data.

In particular, we consider defeasible rules and reasoning, and examine how nonmonotonic (defeasible) reasoning over huge datasets can be performed using massively parallel computational techniques. We adopt the MapReduce framework [5], which is widely used for parallel processing of huge datasets[5].

Our previous work [17] described how defeasible logic with unary predicates can be implemented with MapReduce. In this paper we address the problem for predicates of arbitrary arity. From the applicability perspective, this is a decisive step, as most real-world data require multi-argument predicates. In particular, it opens the possibility of reasoning with semi-structured data, e.g. linked data expressed in RDF, where binary predicates are needed to express properties.

From the technical perspective, multi-argument reasoning with MapReduce turns out to be far more difficult. In [17] fired rules calculation and reasoning are both performed in memory (separately on each unique value), requiring a single MapReduce pass. On the other hand, for the multi-argument case, fired rules calculation (based on joins) and reasoning have to be performed separately, resulting in multiple passes.

In fact, for reasons explained later, our solution works under the requirement that the defeasible theory is stratified. Stratification is a well-known concept used in many areas of knowledge representa-

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

[5] At http://wiki.apache.org/hadoop/PoweredBy, one can see an extensive user list of Hadoop (which is the open-source implementation, of MapReduce framework that we have used); the list includes, among others, IBM, Yahoo!, Facebook and Twitter.

tion, in particular nonmonotonic reasoning, to allow for more efficient reasoning. Regarding applicability, many linked open data domains are stratified (including the well-known LUBM[6] benchmark).

The paper is organized as follows. Section 2 briefly introduces the MapReduce Framework and Defeasible Logic. An algorithm for multi-argument defeasible logic is described in Section 3, while experimental results are presented in Section 4. We conclude in Section 5.

2 Preliminaries

Our implementation is based on two basic components: (a) the MapReduce Framework and (b) the Defeasible Logic. Here we describe briefly the basic notions.

2.1 MapReduce Framework

MapReduce is a framework for parallel processing over huge datasets [5]. 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 parallel. 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 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 *word-count* 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 below.

```
map(Long key, String value):
 // key: position in document (ignored)
 // 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);
```

During 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>.

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

2.2 Defeasible Logic

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

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. There are three types of rules: strict rules, defeasible rules and defeaters represented by a respective arrow $\rightarrow$, $\Rightarrow$ and $\rightsquigarrow$. Strict rules are rules in the classical sense: whenever the premises are indisputable (e.g., facts) then so is the conclusion. Defeasible rules are rules that can be defeated by contrary evidence. Defeaters are rules that cannot be used to draw any conclusions; their only use is to prevent some conclusions.

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 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 we have proved that q is not definitely provable in D.
- $+\partial$q, meaning that q is defeasibly provable in D.
- $-\partial$q, meaning that we have proved that q is not defeasibly provable in D.

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 conditions shown below. 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 [12].

$+\Delta$: 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..i) or
(2.3) $\exists$s $\in$ R[$\sim$q] such that
(2.3.1) $\forall \alpha \in$ A(s): $+\partial\alpha \in$ P(1..i) and
(2.3.2) $\forall$t $\in R_{sd}$[q] either
$\exists \alpha \in$ A(t): $-\partial\alpha \in$ P(1..i) or t $\not\succ$ s

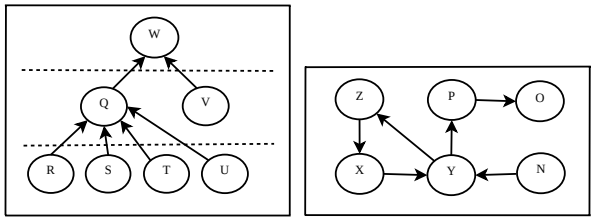
3 Algorithm description

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.



(a) Stratified (b) Non-stratified

Figure 1: Predicate dependency graph

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

r1: R(X,Z), S(Z,Y) $\Rightarrow$ Q(X,Y).
r2: T(X,Z), U(Z,Y) $\Rightarrow \neg$Q(X,Y).
r3: Q(X,Y), V(Y,Z) $\Rightarrow$ W(X,Z).
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: W
Stratum 1: Q, V
Stratum 0: R, S, T, U

Stratified theories are often called decisive in the literature [3].

Proposition 1. *[3] 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's consider for example the facts *R(a,b)*, *S(b,b)*, *T(a,e)*, *U(e,b)* and *V(b,c)*. The *initial pass* on these facts using the aforementioned rule set will create the following output:

<R(a,b), ($+\Delta$,$+\partial$)> <S(b,b), ($+\Delta$,$+\partial$)>
<T(a,e), ($+\Delta$,$+\partial$)> <U(e,b), ($+\Delta$,$+\partial$)>
<V(b,c), ($+\Delta$,$+\partial$)>

No reasoning needs to be performed for the lowest stratum (stratum 0) since these predicates (R,S,T,U) 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 based on techniques used for basic and multi-way join as described in [7]

and [1]. Here we elaborate 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:

$$\text{r1: } R(X,Z), S(Z,Y) \Rightarrow Q(X,Y).$$

The key observation is that relations R and S can be joined on their common argument Z. Based on this observation, during Map operation we emit pairs of the form <Z,(X,R)> for predicate R and <Z,(Y,S)> for predicate S. The idea is to join R and S only for literals that have the same value on argument Z. During $Reduce$ operation we combine R and S producing Q.

In our example, the facts *R(a,b)* and *S(b,b)* will cause Map to emit <b,(a,R)> and <b,(b,S)>. MapReduce framework groups and sorts intermediate pairs passing <b,<(a,R),(b,S)>> to $Reduce$ operation. Finally, at $Reduce$ we combine given values and infer Q(a,b).

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 below. 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.

```
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 +Δ premises 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:

<R(a,b), ($+\Delta$,$+\partial$)> <S(b,b), ($+\Delta$,$+\partial$)>
<T(a,e), ($+\Delta$,$+\partial$)> <U(e,b), ($+\Delta$,$+\partial$)>
<V(b,c), ($+\Delta$,$+\partial$)>

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:

<b, (a,R,$+\Delta$,$+\partial$)> <b, (b,S,$+\Delta$,$+\partial$)>
<e, (a,T,$+\Delta$,$+\partial$)> <e, (b,U,$+\Delta$,$+\partial$)>
<b, (c,V,$+\Delta$,$+\partial$)>

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

<b, <(a,R,$+\Delta$,$+\partial$), (b,S,$+\Delta$,$+\partial$), (c,V,$+\Delta$,$+\partial$)>>
<e, <(a,T,$+\Delta$,$+\partial$), (b,U,$+\Delta$,$+\partial$)>>

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, ([¬,]+∂,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, ([¬,]+Δ,+∂,ruleID)> instead of <firedLiteral, ([¬,]+∂, ruleID)>.

For example during the reduce phase the reducer with key:

b will emit <Q(a,b), ($+\partial$, r1)>
e will emit <Q(a,b), (¬,$+\partial$, r2)>

As we see here, *Q(a,b)* and ¬*Q(a,b)* are computed by different reducers which do not communicate with each other. Thus, none of the two reducers have 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. During the reduce phase we cannot join *Q(a,b)* with *V(b,c)* because we do not have a final conclusion on *Q(a,b)*. Thus, we will not perform reasoning for *W(a,c)* during the second pass, which leads to data loss. However, if another rule (say r4) supporting ¬*W(a,c)* had also fired, then during the second pass, we would have mistakenly inferred ¬*W(a,c)*, leading our knowledge base to inconsistency.

In case of multi-way joins we compute the head of the rule (*firedLiteral*) by performing joins in one or more MapReduce passes as explained in [7] 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, ([¬,]+Δ,+∂,ruleID)> or <firedLiteral, ([¬,]+∂, 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 a duplicate elimination by not emitting pairs for existing conclusions. The pseudo-code for Map and $Reduce$ functions, for stratified rule sets, is depicted below.

```
map(Long key, String value) :
 // key: position in document (irrelevant)
 // value: inferred knowledge/fired rules
 String p = extractLiteral(value);
 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 with +Δ premises then
  Emit(literal, "+Δ, +∂");
 Else If literal is +∂ after reasoning then
  Emit(literal, "+∂");
```

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

<R(a,b), (+Δ,+∂)> <S(b,b), (+Δ,+∂)>
<T(a,e), (+Δ,+∂)> <U(e,b), (+Δ,+∂)>
<V(b,c), (+Δ,+∂)> <Q(a,b), (+∂, r1)>
<¬Q(a,b), (+∂, 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 "¬" in *knowledge* distinguishes p from $\neg p$. The Map function will emit the following pairs:

<R(a,b), (+Δ,+∂)> <S(b,b), (+Δ,+∂)>
<T(a,e), (+Δ,+∂)> <U(e,b), (+Δ,+∂)>
<V(b,c), (+Δ,+∂)> <Q(a,b), (+∂, r1)>
<Q(a,b), (¬, +∂, r2)>

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

<R(a,b), (+Δ,+∂)> <S(b,b), (+Δ,+∂)>
<T(a,e), (+Δ,+∂)> <U(e,b), (+Δ,+∂)>
<V(b,c), (+Δ,+∂)> <Q(a,b), <(+∂, r1), (¬,+∂, 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:

R(a,b), S(b,b), T(a,e), U(e,b), S(b,c), V(b,c) will not emit anything
Q(a,b) will emit <Q(a,b), (+∂)>

Literals *R(a,b), S(b,b), T(a,e), U(e,b), S(b,c)* and *V(b,c)* are known knowledge. For known knowledge a potential duplicate elimination must be performed. We reason simultaneously both for Q(a,b) and ¬Q(a,b). As ¬Q(a,b) is $-\partial$, it does not need to be recorded. Note that duplicate elimination affects the parallelization. This issue is discussed in Section 4.

In case of a highly skewed dataset, first pass may calculate more than once that a certain literal is supported by the same rule. This results, during the second pass, in literals with highly skewed amounts of corresponding knowledge, decreasing the overall parallelization. We address this issue by partially eliminating identical knowledge between map and reduce phases, in an intermediate phase known as the combiner.

3.4 Final remarks

As we see, the approach for multi-argument predicates turns out to be far more difficult, requiring multiple passes compared to the single-pass approach of [17]. Moreover the total number of MapReduce passes is independent of the size of the given input. As mentioned in subsection 3.2, performing reasoning for each stratum separately eliminates data loss and inconsistency, thus out 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 [11], 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 Experimental results

We have implemented our method using Hadoop, and provide experimental results on a cluster, as described below. We have evaluated our system in terms of its ability to handle large data files, its scalability with the number of compute nodes and its scalability with regard to the number of rules in each stratum.

Dataset. The literature in parallel reasoning has traditionally focused on Semantic Web data and monotonic rulesets (e.g. as in [18]). As such, they would not be appropriate for evaluating our system (which supports nonmonotonic logics as well). In the absence of an available benchmark that includes defeasible rules, we based our experiments on manually generated datasets. The generated dataset consists of a set of $+\Delta$ literals. Each literal is represented either as "*predicate(argumentValue)* $+\Delta$" or as "*predicate(argumentValue, argumentValue)* $+\Delta$". In order to simulate a real-world dataset, we used statistics on semantic web data. As described in [9] and [6], semantic web data are highly skewed following zipf distribution. Thus, we used a zipf distribution generator in order to create a dataset resembling real-world datasets. For our experiments, we generated a total of 500 million facts corresponding to 10 GB of data.

Rule set. To the best of our knowledge, there exists no standard defeasible logic rule set to evaluate our approach. For evaluation purposes, taking into consideration rule sets appearing in [11], we created an artificial rule set named **blocking(n)**. In **blocking(n)** there are $n/2$ rules of the form "Q_i(X), R_i(X,Y) $\Rightarrow$ Q_{i+1}(Y)" and $n/2$ of the form "Q_i(X), S_i(X,Y) $\Rightarrow$ $\neg Q_{i+1}$(Y)". Rules supporting Q_{i+1}*(Y)* are superior to rules supporting $\neg Q_{i+1}$*(Y)*, resulting $n/2$ superiority relations. For experimental results we used **blocking(n)** for $n = \{2, 4, 8, 16\}$.

Platform. We have experimented on a cluster of virtual machines on an IBM Cloud, running IBM Hadoop Cluster v1.3, which is compatible with Apache Hadoop v20.2. Each node was equipped with a single CPU core, 1GB of main memory and 55GB of hard disk space. We have scaled the number of nodes from 1 to 16, using a single master node.

Results. Figure 2 shows the scaling properties of our system for 2, 4, 8 and 16 rules[7]. We observe the following: (i) even in our modest

[7] Our experiments were limited to 8 rules for a cluster of 1, 2 or 4 nodes, due to insufficient hard disk space.

setup, the runtime is very short, considering the size of the knowledge base, (ii) our system scales linearly with the number of rules, (iii) our system scales linearly with the number of nodes, i.e., the runtime halves when we double the number of nodes.

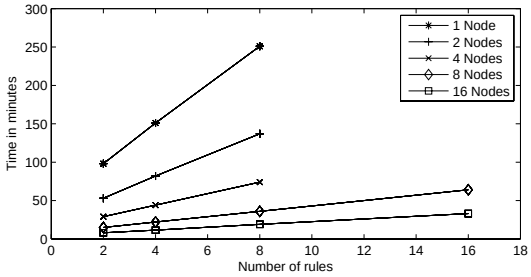


Figure 2: Runtime in minutes for various numbers of rules and nodes.

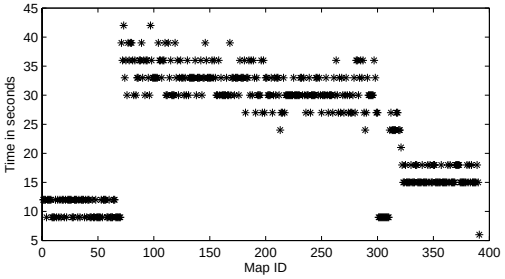


Figure 3: Time in seconds for each map during the second pass

The above show that our system is indeed capable of achieving high performance and scales very well, both with regard to the number of nodes and the number of rules. Nevertheless, to further investigate how our system would perform beyond this, it is critical to examine the load-balancing properties of our algorithm, a major scalability barrier in parallel applications in this domain [9]. Figure 3 shows the load balance between different map tasks during the second pass for 16 nodes on 8 rules. 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 obvious that no such task exists. As described above, during the second pass we filter out literals at Map, which results in a minor work load imbalance (see Figure 3). Finally, our findings show that for highly skewed datasets, the partial elimination of identical knowledge (see Section 3.3), which is often referred to as duplicate elimination, retains the parallelization of our approach for $Reduce$ as well.

It is also worth noting that the communication model of Hadoop is not widely affected by the number of nodes in the cluster. Map operations only use local data (implying very little communication costs). Reduce operations use hash-partitioning to distribute data across nodes based on the keys assigned by the reduce phase. As a result, there is very little locality between data in the Map and the Reduce phase regardless of the number of compute nodes.

5 CONCLUSION

In this paper we extended previous work described in [17] by proposing a method to perform reasoning for multi-argument predicates under the assumption of stratification. Multi-argument predicates complicate significantly the implementation (compared to the one presented in [17] for single-argument predicates), because they require multiple passes. We presented how reasoning can be implemented using the MapReduce framework and provided an experimental evaluation. The results demonstrate that our approach can address reasoning over hundreds of millions of facts.

We consider this to be another step in the research effort towards supporting scalable parallel inconsistency-relevant reasoning. This is important to allow inconsistency-relevant reasoning with huge datasets, that are becoming all the more common (e.g., through the Linked Open Data initiative [4]), as well as to prevent triviality of inference in case of low quality data. Thus, our results could find applications in the context of aggregating, understanding or exploiting the full semantical information of such huge datasets, through their association with customized inference semantics and commonsense reasoning.

Another potential area that may benefit from this work is AI Planning. Given the nonmonotonic nature of planning, in our future work we plan to apply the techniques developed here to help stratify large action databases, in order to leverage massively parallel computation to speed up goal achievement and hence plan construction.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] F. N. Afrati and J. D. Ullman, 'Optimizing joins in a mapreduce environment', in *EDBT*, (2010).
[2] G. Antoniou and F. van Harmelen, *A Semantic Web Primer, 2nd Edition*, The MIT Press, 2 edn., March 2008.
[3] David Billington, 'Defeasible Logic is Stable', *J. Log. Comput.*, **3**(4), 379–400, (1993).
[4] C. Bizer, T. Heath, and T. Berners-Lee, 'Linked Data - The Story So Far', *Int. J. Semantic Web Inf. Syst.*, **5**(3), 1–22, (2009).
[5] Jeffrey Dean and Sanjay Ghemawat, 'MapReduce: simplified data processing on large clusters'.
[6] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, 'Apples and oranges: a comparison of RDF benchmarks and real RDF datasets'.
[7] Florian Fische, 'Investigation & Design for Rule-based Reasoning', Technical report, LarKC, (2010).
[8] E. L. Goodman, E. Jimenez, D. Mizell, S. Al-Saffar, B. Adolf, and D. J. Haglin, 'High-Performance Computing Applied to Semantic Databases', in *ESWC (2)*, pp. 31–45, (2011).
[9] S. Kotoulas, E. Oren, and F. van Harmelen, 'Mind the data skew: distributed inferencing by speeddating in elastic regions', in *WWW*, pp. 531–540, (2010).
[10] S. Kotoulas, F. van Harmelen, and J. Weaver, 'KR and Reasoning on the Semantic Web: Web-Scale Reasoning', (2011).
[11] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller, 'Efficient Defeasible Reasoning Systems', *IJAIT*, **10**, 2001, (2001).
[12] Michael J. Maher, 'Propositional Defeasible Logic has Linear Complexity', *CoRR*, **cs.AI/0405090**, (2004).
[13] J. Maluszynski and A. Szalas, 'Living with Inconsistency and Taming Nonmonotonicity', in *Datalog*, pp. 384–398, (2010).
[14] R. Mutharaju, F. Maier, and P. Hitzler, 'A MapReduce Algorithm for EL+', in *Description Logics*, (2010).
[15] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen, 'Marvin: Distributed reasoning over large-scale Semantic Web data', *J. Web Sem.*, **7**(4), 305–316, (2009).
[16] Y. Roussakis, G. Flouris, and V. Christophides, 'Declarative Repairing Policies for Curated KBs', in *HDMS*, (2011).
[17] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, 'Towards Parallel Nonmonotonic Reasoning with Billions of Facts', in *KR*, (2012).
[18] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal, 'OWL reasoning with webPIE: Calculating the Closure of 100 Billion Triples', in *ESWC (1)*, pp. 213–227, (2010).
[19] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, 'Scalable Distributed Reasoning Using MapReduce', in *ISWC*, pp. 634–649, (2009).