Large scale nonmonotonic reasoning

Ilias Tachmazidis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

This work has been performed at the **University of Crete, School of Sciences and Engineering, Computer Science Department.**

The work is partially supported by the **Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).**

University of Crete
Computer Science Department

Large scale nonmonotonic reasoning

Thesis submitted by
Ilias Tachmazidis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Ilias Tachmazidis

Committee approvals: _____
Dimitris Plexousakis
Professor, Thesis Supervisor

_____
Grigoris Antoniou
Professor, Committee Member

_____
Ioannis Tzitzikas
Assistant Professor, Committee Member

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, July 2012

Abstract

We are witnessing an explosion of available data coming from the Web, government authorities, scientific databases, sensors 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 the vast amounts of data for these applications. In this work, 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. First, we provide a solution for reasoning over single-argument predicates. Subsequently, we extend our approach 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. In particular, we are presenting a scalable method for nonmonotonic rule-based reasoning over Semantic Web Data, using MapReduce. Our experimental results demonstrate that for the single-argument approach, defeasible reasoning with billions of data is performant, and has the potential to scale to trillions of facts. The multi-argument approach has been evaluated with millions of data, proving itself feasible, and having the potential to scale to billions of facts. Finally, our results indicate that nonmonotonic reasoning over RDF 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.

Περίληψη

Είμαστε μάρτυρες μιας έκρηξης διαθέσιμων δεδομένων, προερχόμενων από το Διαδίκτυο, τις κυβερνητικές αρχές, τις επιστημονικές βάσεις δεδομένων, τους αισθητήρες και άλλες πηγές. Τέτοια σύνολα δεδομένων θα μπορούσαν να επωφεληθούν από την εισαγωγή συνόλων από κανόνες οι οποίοι κωδικοποιούν κοινά αποδεκτούς κανόνες ή δεδομένα, ειδικούς κανόνες για ορισμένες εφαρμογές ή πεδία, την γνώση της κοινής λογικής κλπ. Αυτό εγείρει το ερώτημα ως προς το αν, πως, και σε ποιά έκταση οι μέθοδοι της αναπαράστασης γνώσης είναι ικανές να χειριστούν τις τεράστιες ποσότητες δεδομένων για αυτές τις εφαρμογές. Σε αυτήν την εργασία, θεωρούμε συλλογισμούς με ανοχή στην ασυνέπεια υπό την μορφή της αναιρέσιμης συλλογιστικής, και αναλύουμε το πώς ο παραλληλισμός, χρησιμοποιώντας το πλαίσιο εργασίας MapReduce, μπορεί να χρησιμοποιηθεί για συλλογισμούς με αναιρέσιμους κανόνες πάνω σε τεράστια σύνολα δεδομένων. Αρχικά, παρέχουμε μια λύση για συλλογισμούς πάνω σε κατηγορήματα με ένα όρισμα. Στην συνέχεια, επεκτείνουμε την προσέγγισή μας αντιμετωπίζοντας το πρόβλημα για κατηγορήματα πολλών ορισμάτων, υπό την υπόθεση της διαστρωμάτωσης. Η μετακίνηση από κατηγορήματα ενός ορίσματος σε κατηγορήματα πολλών ορισμάτων, είναι ένα αποφασιστικό βήμα προς πρακτικές εφαρμογές, π.χ. συλλογισμοί με Συνδεδεμένα (RDF) Δεδομένα. Συγκεκριμένα, παρουσιάζουμε μια κλιμακούμενη μέθοδο για μη μονότονους συλλογισμούς βασισμένους σε κανόνες, πάνω σε δεδομένα Σημασιολογικού Ιστού, χρησιμοποιώντας MapReduce. Τα πειραματικά μας αποτελέσματα επιδεικνύουν ότι η προσέγγιση για κατηγορήματα ενός ορίσματος είναι αποδοτική για δισεκατομμύρια δεδομένων, και έχει την δυνατότητα να επεκταθεί σε τρισεκατομμύρια δεδομένων. Η προσέγγιση για κατηγορήματα πολλών ορισμάτων, έχει αξιολογηθεί για εκατομμύρια δεδομένων, αποδεικνύοντας ότι είναι εφικτή, έχοντας την δυνατότητα να επεκταθεί σε δισεκατομμύρια δεδομένων. Τέλος, τα αποτελέσματά μας υποδεικνύουν ότι μη μονότονοι συλλογισμοί πάνω σε RDF επιδεικνύουν καλές ιδιότητες επεκτασιμότητας και είναι ικανοί να διαχειριστούν συλλογή δεδομένων συγκριτικής αξιολόγησης που αποτελείται από 1 δισεκατομμύριο τριπλέτες, καθιστώντας το εφάμιλλο με προηγμένες μεθόδους για μονότονες λογικές.

Acknowledgements

First of all, I would like to thank my unofficial supervisor Mr. Grigoris Antoniou, who guided me throughout the postgraduate program. Especially for this thesis, his choices on occurring challenges were decisive. I would like to mention that our collaboration, apart from being extremely pleasant, was particularly fertile since parts of this work were published in well known conferences.

I would like to thank my official supervisor Mr. Dimitris Plexousakis, as he accepted my supervision in a late stage of this thesis.

Of course, the result of this work reflects the assistance of Giorgos Flouris and Spyros Kotoulas. I would like to thank Giorgos for his ideas, advices and patience during the preparation of the publications. This work benefited from the assistance and expertise of Spyros, particularly in the experimental field.

I would like to thank Mr. Ioannis Tzitzikas for his participation in the committee and all the interesting conversations that we had.

I would like to acknowledge the financial support of the Intitute of Computer Science (FORTH-ICS). I would like to thank the staff of the FORTH-ICS for helping me in several ways all these years.

Last, but not least, I would like to thank my family and especially my brother Alexander for his encouragement and precious support during my studies. I would also like to thank my friends and people that are close to me for their company and support.

Contents

1 Introduction **3**
- 1.1 Thesis contribution . . . 5
- 1.2 Thesis outline . . . 5

2 Background **7**
- 2.1 Defeasible Logic . . . 7
  - 2.1.1 Syntax . . . 7
  - 2.1.2 Formal Definition . . . 8
  - 2.1.3 Proof Theory . . . 8
- 2.2 MapReduce Framework . . . 10
  - 2.2.1 Programming Model . . . 10
  - 2.2.2 Running Example . . . 10
  - 2.2.3 Hadoop Framework . . . 12
- 2.3 Resource Description Framework (RDF) . . . 14

3 Single-Argument Predicate Implementation **15**
- 3.1 Algorithm description . . . 15

4 Stratified Multi-Argument Predicate Implementation **19**
- 4.1 Algorithm description . . . 19
  - 4.1.1 Reasoning overview . . . 21
  - 4.1.2 Pass #1: Fired rules calculation . . . 22
  - 4.1.3 Pass #2: Defeasible reasoning . . . 24
  - 4.1.4 Final remarks . . . 26

5 Stratified Multi-Argument Predicate Implementation over RDF **27**
- 5.1 Algorithm description . . . 27
  - 5.1.1 Reasoning overview . . . 27
  - 5.1.2 Pass #1: Fired rules calculation . . . 28
  - 5.1.3 Pass #2: Defeasible reasoning . . . 30

6 Experimental Evaluation **31**
- 6.1 Single-Argument Predicate Evaluation . . . 31
- 6.2 Stratified Multi-Argument Predicate Evaluation . . . 33

6.3 Stratified Multi-Argument Predicate Evaluation over RDF 35

7 Conclusion and Future Work **39**

List of Figures

2.1 Detailed Hadoop MapReduce data flow. 13
2.2 Semantic web stack. 14

4.1 Predicate dependency graph. 20

5.1 Predicate dependency graph. 28

6.1 Runtime in minutes as a function of dataset size, for various numbers of nodes. 32
6.2 Scaled speedup for various dataset sizes. 33
6.3 Runtime in minutes for various numbers of rules and nodes. 34
6.4 Time in seconds for each map during the second pass 34
6.5 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). 37
6.6 Minimum, average and maximum reduce task runtime for each job. 37

Chapter 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 [1]. This yields the need for large and interconnected data, as shown by the Linked Open Data initiative[1] [2].

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 [3] 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 [4, 5] 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 [6, 3, 7, 8], scaling reasoning up to 100 billion triples [9]. 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

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

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 [11]. Monotonic logics such as RDFS cause an explosion of trivial (and often useless derivations), as also identified in [12]. Our work 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 [13], which is widely used for parallel processing of huge datasets[2].

First, a restricted form of defeasible logic is studied: single-argument defeasible logic. A MapReduce algorithm is presented, followed by an extensive experimental evaluation. The findings show that reasoning with billions of facts is possible for a variety of knowledge theories.

Subsequently, we extend our work by addressing 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. For single-argument predicates 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 employed in many areas of knowledge representation for efficiency reasons, e.g., in tractable RDF query answering [14], Description Logics [15, 16, 17] and nonmonotonic formalisms [18], as it has been shown to reduce the computational complexity of various reasoning problems.

Finally, we attempt to evaluate 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

[2]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.

and focussing on performance. A defeasible rule-set for the LUBM[3] 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 [6], we show that our approach performs very well in this respect, distributing data fairly uniformly across MapReduce tasks.

1.1 Thesis contribution

The contribution of this thesis lies in:

- Proposing and evaluating an implementation for single-argument predicates reasoning over huge datasets, using the MapReduce framework (also presented in [19]).
- Proposing and evaluating an implementation for stratified multi-argument predicates reasoning over huge datasets, using the MapReduce framework (also presented in [20]).
- Evaluating the implementation for stratified multi-argument predicates reasoning over RDF data.

1.2 Thesis outline

This thesis is organized as follows. Chapter 2 briefly introduces Defeasible Logic, the MapReduce Framework and RDF. An algorithm for single-argument defeasible logic is described in Chapter 3. Chapter 4 presents an algorithm for stratified multi-argument defeasible logic. Chapter 5 describes how the algorithm for stratified multi-argument defeasible logic can be applied using RDF data. Experimental results, for each implementation, are presented in Chapter 6. Finally, we conclude in Chapter 7 and discuss future work.

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

Chapter 2

Background

2.1 Defeasible Logic

2.1.1 Syntax

A defeasible theory [21, 22, 23] (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:

$$\text{emu(X)} \rightarrow \text{bird(X)}.$$

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "Birds typically fly"; written formally:

$$\text{bird(X)} \Rightarrow \text{flies(X)}.$$

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:

$$\text{heavy(X)} \rightsquigarrow \neg\text{flies(X)}.$$

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

$$\text{r} : \text{bird(X)} \Rightarrow \text{flies(X)}$$

$$\text{r}' : \text{brokenWing(X)} \Rightarrow \neg\text{flies(X)}$$

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.

It is worth noting that, in defeasible logic, priorities are local in the following sense: two rules are considered to be competing with one another only if they have complementary heads. Thus, since the superiority relation is used to resolve conflicts among competing rules, it is only relevant when comparing rules with complementary heads; the information $r > r'$ for rules r, r' without complementary heads may be part of the superiority relation, but has no effect on the proof theory as we will see later.

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

2.1.2 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.

2.1.3 Proof Theory

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

- $+\Delta$q, which is intended to mean that q is definitely provable in D.
- $-\Delta$q, which is intended to mean that we have proved that q is not definitely provable in D.
- $+\partial$q, which is intended to mean that q is defeasibly provable in D.
- $-\partial$q, which is intended to mean 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 following conditions. The conditions are essentially inference rules phrased as conditions on proofs. P(1..i) denotes the initial part of the sequence P of length i.

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

To prove $+\Delta q$ we need to establish a proof for q using facts and strict rules only. This is a deduction in the classical sense. No proofs for the negation of q need to be considered (in contrast to defeasible provability below, where opposing chains of reasoning must be taken into account, too).

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

To prove $-\Delta q$, that is, that q is not definitely provable, q must not be a fact. In addition, we need to establish that every strict rule with head q is known to be inapplicable. Thus, for every such rule r there must be at least one antecedent α for which we have established that α is not definitely provable ($-\Delta\alpha$).

$+\partial$: 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 $t > s$

We can show that q is defeasibly provable, either by showing that q is definitely provable ($+\Delta q$ – see part 1 of the definition for $+\partial$) or by using the defeasible part of D (part 2). For the defeasible part (2) we require that there must be a strict or defeasible rule with head q which can be applied (2.1). But now we need to consider possible attacks, that is, reasoning chains in support of $\sim q$. Thus, we must show that $\sim q$ is not definitely provable (2.2). Additionally, in (2.3) we consider the set of all rules which are not known to be inapplicable and which have head $\sim q$, because, essentially, each such rule s attacks the conclusion q. For q to be provable, each such rule s must be counterattacked by another rule t with head q with the following properties: (i) t must be applicable at this point, and (ii) t must be stronger than s. Thus, each attack on the conclusion q must be counterattacked by a stronger rule.

$-\partial$: 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) $\forall \alpha \in \mathrm{A(s)}$: $+\partial\alpha \in \mathrm{P}(1..\imath)$ and
(2.3.2) $\forall \mathrm{t} \in \mathrm{R}_{sd}[\mathrm{q}]$ either
$\exists \alpha \in \mathrm{A(t)}$: $-\partial\alpha \in \mathrm{P}(1..\imath)$ or $\mathrm{t} \not\succ \mathrm{s}$

To prove that q is not defeasibly provable, we must first establish that it is not definitely provable (1). Then we must establish that it cannot be proven using the defeasible part of the theory (2). There are three possibilities to achieve this: either we have established that none of the (strict and defeasible) rules with head q can be applied (2.1); or ∼q is definitely provable (2.2); or there must be an applicable rule s with head ∼q such that no possibly applicable rule t with head q is superior to s (2.3).

2.2 MapReduce Framework

2.2.1 Programming Model

MapReduce is a framework for parallel processing over huge data sets [13], introduced by Google in 2004. Processing is carried out in 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. Subsequently, 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.

Although MapReduce framework seems restrictive, there are many problems that can be solved by the use of this framework. Some characteristic examples, described in [13], are (1) Distributed Grep, (2) Count of URL Access Frequency, (3) ReverseWeb-Link Graph, (4) Term-Vector per Host, (5) Inverted Index and (6) Distributed Sort.

2.2.2 Running Example

Let us illustrate the *wordcount* example. In this example, we take as input a large number of documents and calculate the frequency of each word. The pseudo-code for the *Map* and *Reduce* functions is depicted in Algorithm 1.

Consider 2 files, each containing lines of text:

File 1:
line 1: “Hello World”
line 2: “Bye World”

File 2:
line 1: “Hello MapReduce Goodbye MapReduce”

Algorithm 1 Wordcount example

```
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 (<w, "1">). Since we have 3 lines in total, 3 *Map* functions will be initiated and run in parallel.

The 1^{st} *Map* function will get as input the following pair:

<0, Hello World>

end emit as output the following pairs:

<Hello, 1>
<World, 1>

The 2^{nd} *Map* function will get as input the following pair:

<11, Bye World>

end emit as output the following pairs:

<Bye, 1>
<World, 1>

The 3^{rd} *Map* function will get as input the following pair:

<0, Hello MapReduce Goodbye MapReduce>

end emit as output the following pairs:

<Hello, 1>
<MapReduce, 1>
<Goodbye, 1>
<MapReduce, 1>

Note that the *key* in input, which is the position in the document, is ignored. Moreover, due to the simplicity of the algorithm, the 3^{rd} *Map* function emits <MapReduce, 1> twice, instead of instantly emitting <MapReduce, 2>.

As mentioned above, the MapReduce framework will group and sort pairs by their key resulting in the following intermediate pairs:

<Bye, 1>
<Goodbye, 1>
<Hello, <1,1>>
<MapReduce, <1,1>>
<World, <1,1>>

The *Reduce* function has to sum up all occurrence values for each word emitting a pair containing the word and the frequency of the word. Since we have 5 groups in total, 5 *Reduce* functions will be initiated and run in parallel. During the reduce phase the reducer with key:

Bye will emit <Bye, 1>
Goodbye will emit <Goodbye, 1>
Hello will emit <Hello, 2>
MapReduce will emit <MapReduce, 2>
World will emit <World, 2>

The final result is the output of all *Reduce* functions, namely:

<Bye, 1>
<Goodbye, 1>
<Hello, 2>
<MapReduce, 2>
<World, 2>

2.2.3 Hadoop Framework

In this work we adopt the Hadoop[1] framework, a Java-based open-source implementation of the MapReduce framework. Hadoop is a project of *The Apache Software Foundation*, with an extensive user list including companies like IBM, Yahoo!, Facebook and Twitter[2].

A MapReduce program execution in Hadoop is called a *job*. Hadoop is based on the master/slave architecture. Thus, there is one node, called master, coordinating the rest nodes (called slaves). Master node runs two demons, the first called "JobTracker", which is responsible for the assignment and coordination of jobs, and the second called "Namenode", which manages the file system namespace and regulates access to files. Each slave node runs two demons as well, the first

[1] http://hadoop.apache.org/mapreduce/
[2] `http://wiki.apache.org/hadoop/PoweredBy`

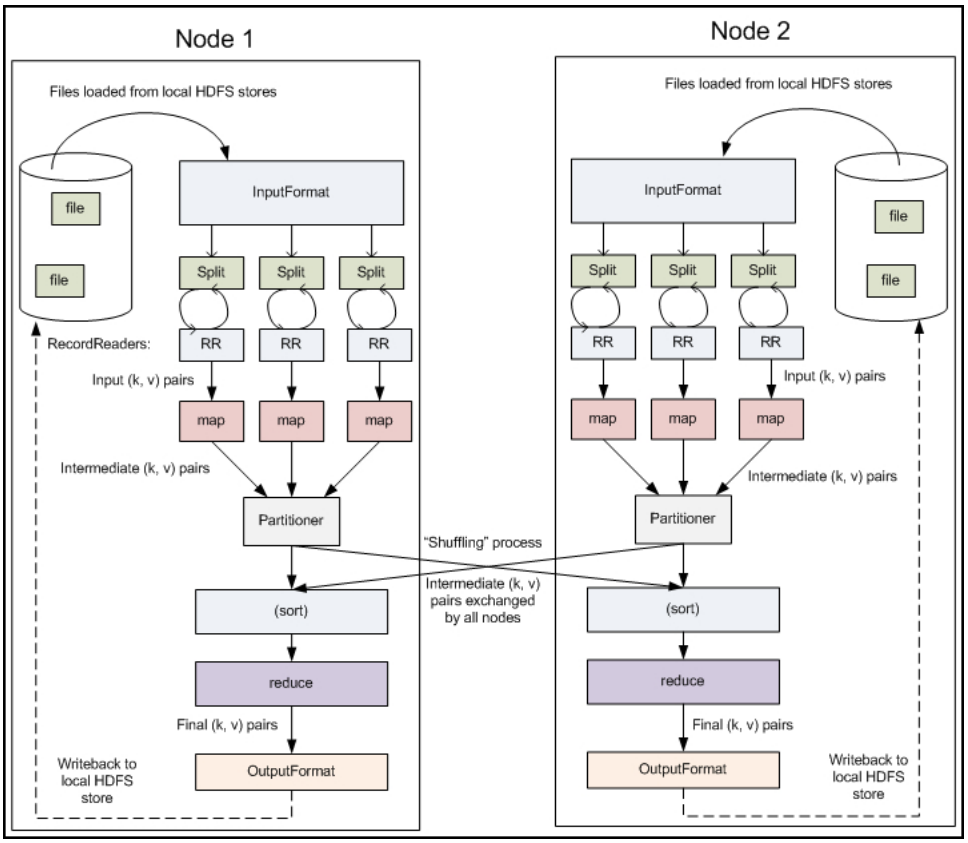


Figure 2.1: Detailed Hadoop MapReduce data flow.

called "TaskTracker", which accepts and runs parts of the computation of a job, and the second called "Datanode" which manages storage attached to the node.

Figure 2.1[3] illustrates an overview of the Hadoop MapReduce data flow for 2 nodes. Nonetheless, this structure is followed independently of the number of nodes.

Input and output files are stored in the Hadoop Distributed File System (HDFS). HDFS is built for the purposes of Hadoop and it uniformly distributes files across all nodes in the cloud (the term *node* refers to a computer in the cloud).

At the beginning of a job, Hadoop reads the input files from HDFS and splits the input into chunks. Each chunk is assigned to a node. However, input must be fed to each *Map* function in the form of key/value pairs. The transformation of the input into key/value pairs is performed by RecordReaders.

Each map operation runs, in parallel with the others, the user-defined *Map* function and emits zero or more intermediate pairs. All these intermediate pairs are redirected to a node by the *Partitioner*, for the reduce phase, according to a hash

[3]Taken from the Yahoo! tutorial `http://developer.yahoo.com/hadoop/tutorial/module4.html`

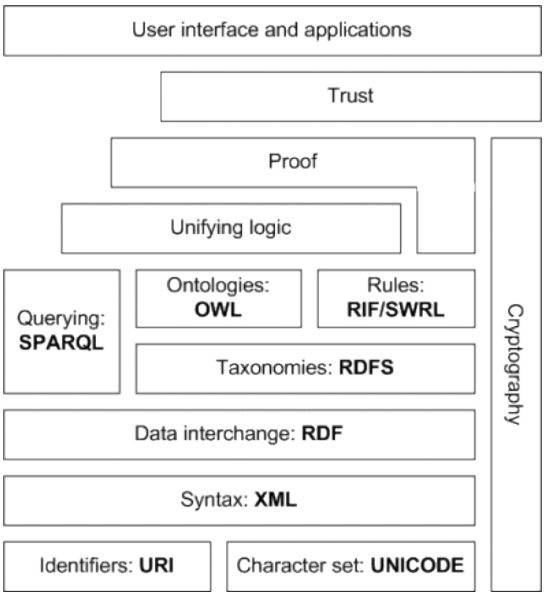


Figure 2.2: Semantic web stack.

function. The hash function of the Partitioner is defined by default as the hash of the key and modulo the number of reduce operations. Once all intermediate pairs are shuffled, each node sorts locally stored pairs by their key, generating key/list(value) groups. A *Reduce* function is applied, in parallel with the others, for each key/list(value) group, emitting zero or more key/value pairs. The output of all reduce operations is the final output of the job and it is stored in HDFS.

2.3 Resource Description Framework (RDF)

Resource Description Framework (RDF) is an official W3C Recommendation for Semantic Web data models [24]. As we see in Figure 2.2, RDF is used for data interchange. The information is encoded in *triples* of the form "subject *predicate* object". The *subject* is an RDF URI reference or a blank node[4], the *predicate* is an RDF URI reference, and the *object* is an RDF URI reference, a literal[5] or a blank node. Intuitively, *predicate* denotes a relationship between *subject* and *object*.

Consider the following sentence:

"This thesis is authored by Ilias Tachmazidis."

It can be expressed as an RDF triple of the form:

<http://www.csd.uoc.gr/thesis> <http://www.csd.uoc.gr/authoredBy> "Ilias Tachmazidis"

[4]http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#dfn-blank-node

[5]http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#dfn-literal. The term *literal*, that is mentioned here, should not be confused with the term *literal* for defeasible logic.

Chapter 3

Single-Argument Predicate Implementation

3.1 Algorithm description

The implementation for single-argument predicates is based on the combination of defeasible logic with MapReduce. In order to achieve this combination we have to take into consideration the characteristics of each component.

As a running example, let us consider the following rule set:

r1 : bird(X) $\rightarrow$ animal(X)

r2 : bird(X) $\Rightarrow$ flies(X)

r3 : brokenWing(X) $\Rightarrow \neg$flies(X)

r3 $>$ r2

In this simple example we try to decide whether something is an animal and whether it is flying or not. Given the facts *bird(eagle)* and *brokenWing(eagle)*, as well as the superiority relation, we conclude that *animal(eagle)* and $\neg$*flies(eagle)*.

Taking into account the fact that all predicates have only one argument, we can group together facts with the same argument value (using Map) and perform reasoning for each value separately (using Reduce). Pseudo-code for *Map* and *Reduce* functions is depicted in Algorithm 2. Equivalently, we can view this process as performing reasoning on the rule set:

r1 : bird $\rightarrow$ animal

r2 : bird $\Rightarrow$ flies

r3 : brokenWing $\Rightarrow \neg$flies

Algorithm 2 single-argument inference

```
map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: document line (a fact)
  String argumentValue = extractArgumentValue(value);
  String predicate = extractPredicate(value);
  EmitIntermediate(argumentValue, predicate);

reduce(String key, Iterator values) :
  // key: argument value
  // values : list of predicates (facts)
  List listOfFacts;
  Reasoner reasoner = Reasoner.getCopy();
  for each v in values
    listOfFacts.add(v);
  reasoner.Reason(listOfFacts);
  Emit(key , reasoner.getResults());
```

r3 > r2

for each unique argument value.

As far as MapReduce is concerned, the *Map* function reads facts of the form *predicate(argumentValue)* and emits pairs of the form <argumentValue, predicate>.

Given the facts: *bird(eagle)*, *bird(owl)*, *bird(pigeon)*, *brokenWing(eagle)* and *brokenWing(owl)*, the *Map* function will emit the following pairs:

<eagle, bird>

<owl, bird>

<pigeon, bird>

<eagle, brokenWing>

<owl, brokenWing>

Then, reasoning is performed for each argument value (e.g., eagle, pigeon etc) separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

<eagle, <bird, brokenWing>>

<owl, <bird, brokenWing>>

<pigeon, <bird>>

Reasoning is then performed during the reduce phase for each argument value in isolation, using the second rule set presented earlier (propositional form). For each *Reduce* function, a copy of reasoner (described later on) gets as input a list of predicates and performs reasoning deriving and emitting new data. When all reduces are completed, the whole process is completed guaranteeing that every possible new data is inferred.

Returning to our example, the bullets below show the reasoning tasks that need to be performed. Note that each of these reasoning tasks can be performed in parallel with the others.

- *eagle* having *bird* and *brokenWing* as facts, deriving *animal(eagle)* and ¬*flies(eagle)*
- *owl* having *bird* and *brokenWing* as facts, deriving *animal(owl)* and ¬*flies(owl)*
- *pigeon* having *bird* as fact, deriving *animal(pigeon)* and *flies(pigeon)*.

For the purpose of conclusion derivation, we implemented a reasoner based on a variation of algorithm for propositional reasoning, described in [25]. Prior to any *Reduce* function is applied, given rule set must be parsed initializing indexes and data structures required for reasoning. Since the algorithm for the reasoner is well defined in [25], we will not focus on implementation details of the reasoner, but, we will explain all the functions used in Algorithm 2.

Each *Reduce* function has to perform, in parallel, reasoning on the initial state of the reasoner. Thus, we use *Reasoner.getCopy()*, which provides a copy of the initialized reasoner. Subsequently, *reasoner.Reason(listOfFacts)* performs reasoning on each copy. In order to perform reasoning, *reasoner.Reason(listOfFacts)* gets as input the corresponding list of predicates (*listOfFacts*). Derived data are stored internally by each copy of the reasoner. The extraction of the derived data is performed by the *reasoner.getResults()*.

The algorithm for single-argument predicates is sound and complete since it performs reasoning using every given fact. This data partitioning does not alter resulting conclusions since facts with different argument values cannot produce conflicting literals and cannot be combined to reach new conclusions. Moreover, the reasoner is designed to derive all possible conclusions for each unique value. Thus, maximal and valid data derivation is assured.

Chapter 4

Stratified Multi-Argument Predicate Implementation

4.1 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.

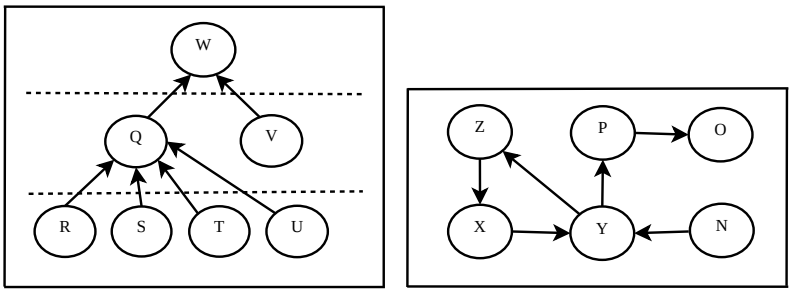
The dependencies between predicates can be represented using a *predicate dependency graph*. For a given rule set, the *predicate dependency graph* is a directed graph whose:

- 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 4.1a depicts the predicate dependency graph of a stratified rule set, while Figure 4.1b 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:



(a) Stratified (b) Non-stratified

Figure 4.1: Predicate dependency graph.

r1: R(X,Z), S(Z,Y) ⇒ Q(X,Y).
r2: T(X,Z), U(Z,Y) ⇒ ¬Q(X,Y).
r3: Q(X,Y), V(Y,Z) ⇒ W(X,Z).
r1 > r2.

The predicate dependency graph for the above rule set is depicted in Figure 4.1a. 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 4.1a 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 [18].

Proposition 1 *[18] 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

Algorithm 3 Overall process

```
initial_pass();
For each stratum from 1 to N
  pass1_calculateFiredRules();
  pass2_performDefeasibleReasoning();
```

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.

4.1.1 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. Pseudo-code for the overall process is depicted in Algorithm 3. In order to perform defeasible reasoning we have to run two passes for each stratum. The first pass computes which rules can fire (*pass1_ calculateFiredRules()*). The second pass performs the actual reasoning and computes for each literal if it is definitely or defeasibly provable (*pass2_ performDefeasibleReasoning()*). The reasons for both decisions (reasoning sequence and two passes per stratum) are explained in the end of the next subsection.

4.1.2 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 [26, 27]. 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:

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 in Algorithm 4. *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.

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,$+\Delta$,$+\partial$)>

Algorithm 4 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");
```

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

<b, <(a,R,+Δ,+∂), (b,S,+Δ,+∂)>>
<e, <(a,T,+Δ,+∂), (b,U,+Δ,+∂)>>

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. 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 +Δ conclusions inference. If all premises are +Δ 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), (+∂, r1)>
e will emit <Q(a,b), (¬,+∂, 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. We will attempt to perform reasoning for all the strata simultaneously. On the one hand, we cannot

join *Q(a,b)* with *V(b,c)* prior to the second pass 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. On the other hand, if another rule (say r4) supporting $\neg W(a,c)$ had also fired, then during the second pass, we would have mistakenly inferred $\neg W(a,c)$, leading our knowledge base to inconsistency.

In case of multi-way joins we compute fired rules by performing joins in one or more MapReduce passes, as explained in [26, 27]. Consider the following rule:

r: A(X,Y), B(Y,V), C(V,Z) $\Rightarrow$ D(X,Z).

A simple way to compute *r* is to first join *A(X,Y)* and *B(Y,V)* on *Y* in a MapReduce pass, producing an intermediate literal, say *AB(X,V)*. Then, in a second MapReduce pass, we join *AB(X,V)* with *C(V,Z)* on *V* producing the *D(X,Z)*, which is the *firedLiteral*. As above, for each fired rule, we must take into consideration the type of the rule (strict rule, defeasible rule and defeater) and whether all the premises are $+\Delta$ or not. Finally, the format of the output remains the same (<firedLiteral, ([¬,]$+\Delta$,$+\partial$,ruleID)> or <firedLiteral, ([¬,]$+\partial$, ruleID)>).

4.1.3 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 in Algorithm 5.

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

<R(a,b), ($+\Delta$,$+\partial$)>
<S(b,b), ($+\Delta$,$+\partial$)>
<T(a,e), ($+\Delta$,$+\partial$)>
<U(e,b), ($+\Delta$,$+\partial$)>
<V(b,c), ($+\Delta$,$+\partial$)>
<Q(a,b), ($+\partial$, r1)>
<Q(a,b), (¬, $+\partial$, 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 "¬" in *knowledge* distinguishes *p* from $\neg p$. However, we must filter out, from the input, literals that do not belong to the stratum that we currently perform reasoning for. The *Map* function will emit the following pairs:

<V(b,c), ($+\Delta$,$+\partial$)>
<Q(a,b), ($+\partial$, r1)>
<Q(a,b), (¬, $+\partial$, r2)>

Algorithm 5 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:

<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 defeasible reasoning for each literal.

During the reduce phase the reducer with key:

V(b,c) will not emit anything
Q(a,b) will emit <Q(a,b), (+∂)>

Literal *V(b,c)* is 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 filtering out

literals that are not needed for defeasible reasoning of the current stratum, affects the parallelization during the map phase. This issue is discussed in Section 6.2.

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.

4.1.4 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 for single-argument predicates. Moreover, the total number of MapReduce passes is independent of the size of the given input. As mentioned in subsection 4.1.2, 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 [28], 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..$\imath$)" (in $-\Delta$ provability) and "$\forall$r $\in$ R$_{sd}$[q] $\exists\alpha \in$ A(r): $-\partial\alpha \in$ P(1..$\imath$)" (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. We need a more efficient representation of our knowledge base, since the proposed knowledge representation in [28] comes with limitations. Consider storing the cartesian product of all the values of the arguments X, Y, Z for *Q(X,Y,Z)*. After a certain point, the storage of the knowledge base is not feasible even for secondary storage devices, due to the enormous number of available literals.

Chapter 5

Stratified Multi-Argument Predicate Implementation over RDF

5.1 Algorithm description

In this section, we provide a running example for stratified multi-argument implementation over RDF data. We perform defeasible reasoning as it is defined in Section 4.1. Here, each RDF triple "Subject *Predicate* Object" can also be represented as a literal of the form *Predicate(Subject,Object)*, and will be (at some points) referred to as a literal. Nonetheless, in this section we use the RDF triple form instead of the corresponding literal form, in order to provide the intuition behind reasoning over RDF data.

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 5.1. The various strata of the above rule set, are as follows:

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

5.1.1 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, $(+\Delta,+\partial)$>.

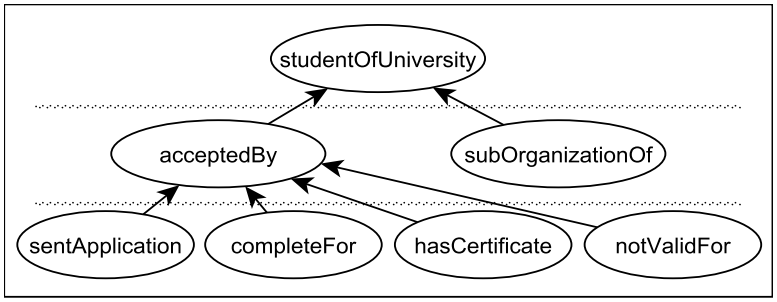


Figure 5.1: Predicate dependency graph.

Now let's 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 described in Section 4.1, for a hierarchy of N strata we have to perform $N-1$ times the procedure described below.

5.1.2 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 [26, 27]. Specifically, for joins over RDF data one can be advised by [29]. 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. The *Map* operation, given <John *sentApplication* App, $(+\Delta,+\partial)$> and <App *completeFor* Dep, $(+\Delta,+\partial)$> as input, emits pairs of the form <App, (John, *sentApplication*, $+\Delta,+\partial$)> for predicate *sentApplication* and <App, (Dep, *completeFor*, $+\Delta,+\partial$)> for predicate *completeFor*. The idea is to join *sentApplication* and *completeFor* only for literals that have the

same value on argument A. The MapReduce framework groups and sorts intermediate pairs passing <App, <(John, *sentApplication*, $+\Delta$,$+\partial$), (Dep, *completeFor*, $+\Delta$,$+\partial$)>> to the *Reduce* operation. During the *Reduce* operation we combine *sentApplication* and *completeFor* producing <John *acceptedBy* Dep, ($+\Delta$,$+\partial$)>.

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 the reduce phase the reducer with key:

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

In case of multi-way joins we compute fired rules by performing joins in one or more MapReduce passes as explained in [26, 27, 29]. Consider the following rule:

r: X *predicateA* Y, Y *predicateB* V, V *predicateC* Z $\Rightarrow$ X *predicateD* Z.

A simple way to compute r is to first join "X *predicateA* Y" and "Y *predicateB* V" on Y in a MapReduce pass, producing an intermediate literal, say "X *predicateAB* V". Then, in a second MapReduce pass, we join "X *predicateAB* V" with "V *predicateC* Z" on V producing the "X *predicateD* Z", which is the *firedLiteral*. For each fired rule, we must take into consideration the type of the rule (strict rule, defeasible rule and defeater) and whether all the premises are $+\Delta$ or not. Finally, the format of the output is either (<firedLiteral, ([$\neg$,]$+\Delta$,$+\partial$,ruleID)> or <firedLiteral, ([$\neg$,]$+\partial$, ruleID)>).

5.1.3 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 5.

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 $\neg p$. However, we must filter out, from the input, literals that do not belong to the stratum that we currently perform reasoning for. The *Map* function will emit the following pairs:

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

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, $(+\partial)$>

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". Finally, as "John $\neg$*acceptedBy* Dep" is $-\partial$, it does not need to be recorded.

Chapter 6

Experimental Evaluation

6.1 Single-Argument Predicate Evaluation

We implemented the algorithm for single-argument predicates in the Hadoop MapReduce framework, version 0.20. We performed experiments on a cluster with 16 IBM System x iDataPlex nodes, using a Gigabit Ethernet interconnect. Each node was equipped with dual Intel Xeon Westmere 6-core processors, 128GB RAM and a single 1TB SATA hard drive.

Dataset. Due to no available benchmark, we generated our data set manually. In order to store facts directly to Hadoop Distributed File System (HDFS), facts were generated using the MapReduce framework. We created a set of files consisting of generated pairs of the form <argumentValue, predicate>, with each pair corresponding to a unique fact. Finally, considering storage space, 1 billion facts correspond to 10 GB of data.

Rule set. To the best of our knowledge, there exist no standard defeasible logic rule set to evaluate our approach. For this reason, we decided to use synthetic rule sets, namely the artificial rule set **teams(n)** appearing in [28]. In **teams(n)** every literal is disputed, with $2^{(2*i)+1}$ rules of the form $a_{i+1} \Rightarrow a_i$ and $2^{(2*i)+1}$ rules of the form $a_{i+1} \Rightarrow \neg a_i$, for $0 \leq i \leq n$. The rules for a_i are superior to the rules for $\neg a_i$, resulting in $2^{(2*i)+1}$ superiority relations, for $0 \leq i \leq n$. For our experiments, we generated a **teams(n)** rule set for $n = 1$, which resulted in 20 defeasible rules, 10 superiority relations, 20 predicates appearing in the body of rules and 5 literals for conclusion derivation. This particular rule set was chosen because it is the only known benchmark for defeasible logics that involves "attacks".

Evaluation settings. We evaluated our system in terms of the following parameters:

- **Runtime**, as the time required to calculate the inferential closure of the input, in minutes.
- **Number of nodes** performing the computation in parallel.
- **Dataset size**, expressed in the number of facts in the input.

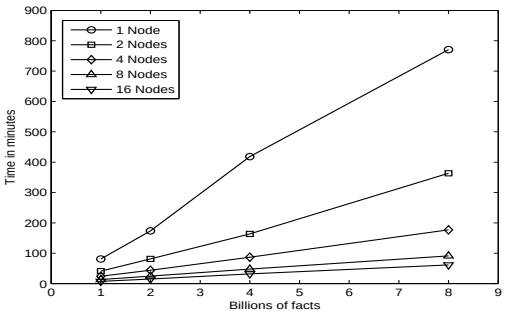


Figure 6.1: Runtime in minutes as a function of dataset size, for various numbers of nodes.

- **Scaled speedup**, defined as $s = \frac{runtime_{1node}}{runtime_{Nnodes} * N}$, where $runtime_{1node}$ is the required run time for one node, N is the number of nodes and $runtime_{Nnodes}$ is the required run time for N nodes. It is a commonly used metric in parallel processing to measure how a system scales as the number of nodes increases. A system is said to scale sublinearly, superlinearly and linearly when $s < 1$, $s > 1$ and $s \simeq 1$ respectively.

Results. Figure 6.1 shows the runtime plotted against the size of the dataset, for various numbers of processing nodes and Figure 6.2 shows the scaled speedup for increasing number of nodes, and for various dataset sizes. Our results indicate the following:

- Our system easily scales to several billions of facts, even for a single node. In fact, we see no indication of the throughput decreasing as the size of the input increases.
- Our implementation demonstrates very high throughput (about 2,2 million facts per second) and is in league with state-of-the-art methods for monotonic logics [9].
- Our system scales fairly linearly with the number of nodes. The loss in terms of scaled speedup for larger numbers of nodes and small datasets is attributed to platform overhead. Namely, starting a computational job in Hadoop incurs a significant computational overhead.
- In some cases, our system scales superlinearly. This is attributed to being able to store a larger part of the data in RAM. Although MapReduce relies on the hard drives for data transfer, the operating system uses RAM to improve disk access time and throughput, which explains the improved performance at some points.

In general, we attribute the demonstrated scalability to: (a) the limited communication required in our model, (b) the carefully designed load-balancing attributes

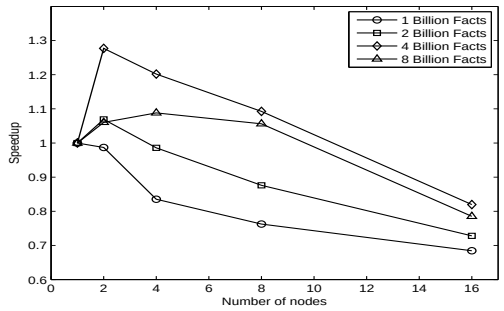


Figure 6.2: Scaled speedup for various dataset sizes.

of our algorithm, and, (c) the efficiency of Hadoop in handling large data volumes. To our best knowledge, this is the first work addressing nonmonotonic reasoning, for single-argument predicates, using mass parallelization techniques. Thus, we were unable to compare our findings with related work.

6.2 Stratified Multi-Argument Predicate Evaluation

We implemented our method using Hadoop, and provide experimental results on a cluster, as described below. We 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. 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 [7, 30], 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 [28], we created an artificial rule set named **blocking(n)**. In **blocking(n)** there are $n/2$ rules of the form "$\mathrm{Q}_i(\mathrm{X}), \mathrm{R}_i(\mathrm{X,Y}) \Rightarrow \mathrm{Q}_{i+1}(\mathrm{Y})$" and $n/2$ of the form "$\mathrm{Q}_i(\mathrm{X}), \mathrm{S}_i(\mathrm{X,Y}) \Rightarrow \neg\mathrm{Q}_{i+1}(\mathrm{Y})$". 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 experimented on a cluster of virtual machines on an IBM Cloud, running IBM Hadoop Cluster v1.3, which is compatible with Apache Hadoop v0.20.2. Each node was equipped with a single CPU core, 1GB of main memory and 55GB of hard disk space. We scaled the number of nodes from 1 to 16,

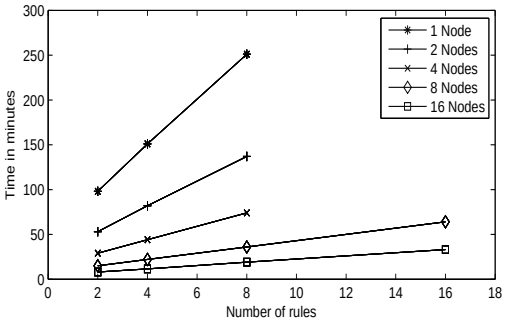


Figure 6.3: Runtime in minutes for various numbers of rules and nodes.

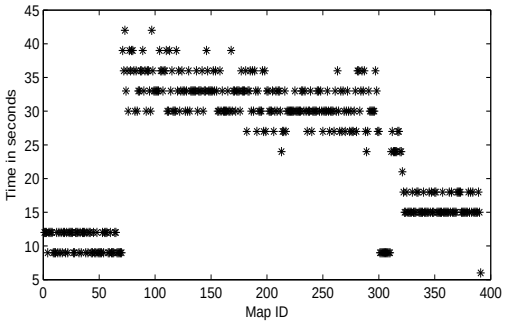


Figure 6.4: Time in seconds for each map during the second pass

using a single master node.

Results. Figure 6.3 shows the scaling properties of our system for 2, 4, 8 and 16 rules[1]. We observe the following: (i) even in our modest 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.

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 [7]. Figure 6.4 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 in Section 4.1.3, during the second pass we filter out literals at *Map*, which results in a minor work load imbalance (see Figure 6.4). Finally, our findings show that for highly skewed

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

datasets, the partial elimination of identical knowledge (see Section 4.1.3), 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.

6.3 Stratified Multi-Argument Predicate Evaluation over RDF

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 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 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 [7]. Thus, we 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.

Data set. We 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.

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 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.
r12 > r11.
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.
r14 > r13.
r15: X worksFor D ⇒ X canBecomeHeadOf D.
r16: X worksFor D, Z headOf D, X commonResearchInterests Z ⇒ X ¬canBecomeHeadOf D.
r16 > r15.
r17: Y teaches X ⇒ X suggestAdvisor Y.
r18: Y teaches X, X hasAdvisor Z ⇝ X ¬suggestAdvisor Y.
r18 > r17.

MapReduce jobs description. We need 8 jobs in order to perform reasoning on the above rule set. The 1^{st} job is the *initial pass* described in Section 5.1 (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 2^{nd} job computes rules r4-r6. During the 3^{rd} job we perform duplicate elimination, since r4-r6 are strict rules. We compute rules r7-r14 during the 4^{th} job, while reasoning on them is performed during the 5^{th} job. Jobs 6 and 7 compute rules r15-r18. Finally, during the 8^{th} job we perform reasoning on r15-r18, finishing the whole procedure.

Platform. Our experiments were performed on a 40-core IBM x3850 server connected to a XIV Storage Area network (SAN), using a 10Gbps storage switch. We 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 [31]. We used a number of Mappers and Reducers equal to the number of cores in the system (i.e. 40).

Results. Figure 6.5 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

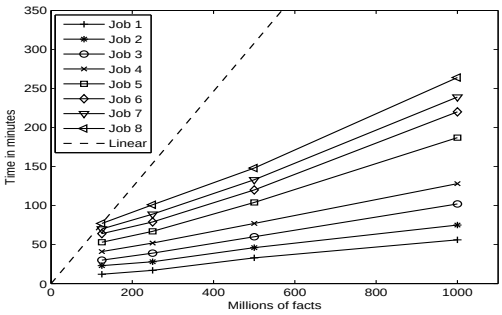
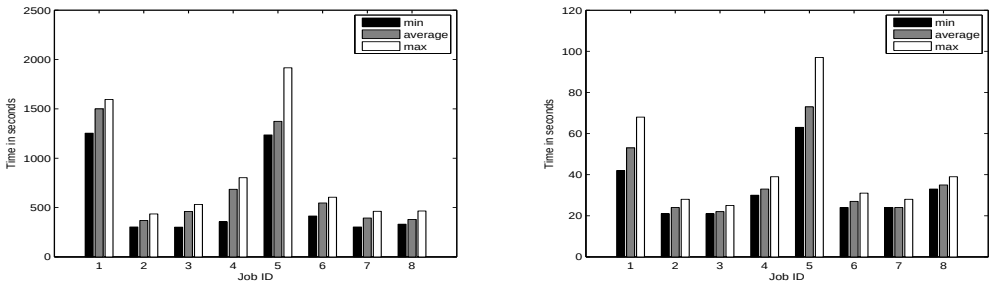


Figure 6.5: Runtime in minutes for various datasets, and projected linear scalability. Job runtimes are stacked (i.e. runtime for Job 8 includes the runtimes for Jobs 1-7).



(a) 40 reduce tasks

(b) 400 reduce tasks

Figure 6.6: Minimum, average and maximum reduce task runtime for each job.

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 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 launched an experiment where we 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 6.6 shows the load balance between different reduce tasks, for 1 billion triples and 40 (Figure 6.6a) or 400 (Figure 6.6b) 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 obvious that no such task exists.

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 [2]. WebPIE [9], 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.

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

Chapter 7

Conclusion and Future Work

This work is the first to explore the feasibility of nonmonotonic reasoning over huge data sets. We focused on simple nonmonotonic reasoning in the form of defeasible logic. We described how defeasible logic can be implemented in the MapReduce framework. We first focused on the case of reasoning with defeasible logic rules containing only single-argument predicates, and provided experimental evaluation for this case. Our results are very encouraging, and demonstrate that one can handle billions of facts using our approach.

Subsequently, we extended our work 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 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.

Thereafter, 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 real 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 run extensive experiments to test the efficiency of multi-argument defeasible logic which, as we explained, needs multiple passes in MapReduce, as well as to study the effect of introducing increasingly complex defeasible rule sets, like those used in [28]. Our expectation is that this case will also turn out to be fully feasible.

In the longer term, we intend to apply the MapReduce framework to more complex knowledge representation methods, such as Answer-Set programming [32] and ontology dynamics (including evolution [33], diagnosis, and repair [1]) 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.

Another potential area that may benefit from this work is AI Planning. Here the most competitive modern planning engines work by first grounding planning operators (which in general are made up of multi-argument predicates) into thousands of ground operators called actions. The pre- and post-condition structure of these actions gives rise to planning dependency graphs also manifested as *causal graphs* [34]. 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.

Bibliography

[1] Y. Roussakis, G. Flouris, and V. Christophides, "Declarative Repairing Policies for Curated KBs," in *HDMS*, 2011.

[2] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009. [Online]. Available: http://eprints.ecs.soton.ac.uk/21285/

[3] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, "Scalable Distributed Reasoning Using MapReduce," in *ISWC*, ser. Lecture Notes in Computer Science, A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, Eds., vol. 5823. Springer, 2009, pp. 634–649.

[4] G. Antoniou and F. van Harmelen, *A Semantic Web Primer, 2nd Edition*, 2nd ed. The MIT Press, March 2008.

[5] J. Maluszynski and A. Szalas, "Living with Inconsistency and Taming Nonmonotonicity," in *Datalog*, 2010, pp. 384–398.

[6] 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.*, vol. 7, no. 4, pp. 305–316, 2009.

[7] S. Kotoulas, E. Oren, and F. van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *WWW*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 531–540.

[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)*, 2011, pp. 31–45.

[9] 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)*, ser. Lecture Notes in Computer Science, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds., vol. 6088. Springer, 2010, pp. 213–227.

[10] S. Kotoulas, F. van Harmelen, and J. Weaver, "KR and Reasoning on the Semantic Web: Web-Scale Reasoning," in *Handbook of Semantic Web Technologies*, J. Domingue, D. Fensel, and J. A. Hendler, Eds. Springer Berlin Heidelberg, 2011, pp. 441–466, 10.1007/978-3-540-92913-0_11. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92913-0_11

[11] R. Mutharaju, F. Maier, and P. Hitzler, "A MapReduce Algorithm for EL+," in *Description Logics*, ser. CEUR Workshop Proceedings, V. Haarslev, D. Toman, and G. E. Weddell, Eds., vol. 573. CEUR-WS.org, 2010.

[12] A. Hogan, A. Harth, and A. Polleres, "Scalable Authoritative OWL Reasoning for the Web," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 49–90, 2009.

[13] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[14] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen, "Containment and Minimization of RDF(S) Query Patterns," in *ISWC-05*, 2005.

[15] F. Baader and R. Kosters, "Nonstandard Inferences in Description Logics: The Story So Far," *Mathematical Problems from Applied Logic I, volume 4 of International Mathematical Series*, 2006.

[16] C. Haase and C. Lutz, "Complexity of Subsumption in the EL Family of Description Logics: Acyclic and Cyclic TBoxes," in *ECAI-08*, 2008, pp. 25–29.

[17] B. Nebel, "Terminological Reasoning is Inherently Intractable," *Artificial Intelligence*, vol. 43, pp. 235–249, 1990.

[18] D. Billington, "Defeasible Logic is Stable," *J. Log. Comput.*, vol. 3, no. 4, pp. 379–400, 1993.

[19] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, "Towards Parallel Nonmonotonic Reasoning with Billions of Facts," in *KR-12*, 2012.

[20] I. Tachmazidis, G. Antoniou, G. Flouris, S. Kotoulas, and L. McCluskey, "Large-scale Parallel Stratified Defeasible Reasoning," in *ECAI-12*, 2012.

[21] D. Nute, "Defeasible Logic," in *Handbook of Logic in Artificial Intelligence and Logic Programming-Nonmonotonic Reasoning and Uncertain Reasoning(Volume 3)*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Oxford: Clarendon Press, 1994, pp. 353–395.

[22] G. Antoniou and M.-A. Williams, *Nonmonotonic reasoning*. MIT Press, 1997.

[23] G. Antoniou and M. Maher, "Embedding defeasible logic into logic programs," in *In Proc. ICLP 2002*. Springer, 2002, pp. 393–404.

[24] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004.

[25] M. J. Maher, "Propositional Defeasible Logic has Linear Complexity," *CoRR*, vol. cs.AI/0405090, 2004.

[26] F. Fische, "Investigation & Design for Rule-based Reasoning," LarKC, Tech. Rep., 2010.

[27] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*, ser. EDBT '10. New York, NY, USA: ACM, 2010, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/1739041.1739056

[28] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller, "Efficient Defeasible Reasoning Systems," *IJAIT*, vol. 10, p. 2001, 2001.

[29] U. J., "RDFS/OWL reasoning using the MapReduce framework," Master's thesis, Vrije Universiteit, 2009.

[30] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, "Apples and oranges: a comparison of RDF benchmarks and real RDF datasets," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989340

[31] K. B. R. Vernica, A. Balmin and V. Ercegovac, "Adaptive Mapreduce using Situation-Aware Mappers," in *EDBT*.

[32] M. Gelfond, "Chapter 7 answer sets," in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, V. L. F. van Harmelen and B. Porter, Eds. Elsevier, 2008, vol. 3, pp. 285 – 316. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574652607030076

[33] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides, "A Formal Approach for RDF/S Ontology Evolution," in *ECAI*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2008, pp. 70–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=1567281.1567301

[34] M. Helmert, "The Fast Downward Planning System," *J. Artif. Intell. Res. (JAIR)*, vol. 26, pp. 191–246, 2006.