



# University of HUDDERSFIELD

## University of Huddersfield Repository

Tachmazidis, Ilias

Large-scale Reasoning with Nonmonotonic and Imperfect Knowledge Through Mass Parallelization

### Original Citation

Tachmazidis, Ilias (2015) Large-scale Reasoning with Nonmonotonic and Imperfect Knowledge Through Mass Parallelization. Doctoral thesis, University of Huddersfield.

This version is available at <http://eprints.hud.ac.uk/id/eprint/27005/>

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: [E.mailbox@hud.ac.uk](mailto:E.mailbox@hud.ac.uk).

<http://eprints.hud.ac.uk/>

# Large-scale Reasoning with Nonmonotonic and Imperfect Knowledge Through Mass Parallelization

Ilias Tachmazidis

Thesis Supervisor: Prof. *Grigoris Antoniou*

Thesis Co-supervisor: Prof. *Thomas Leo McCluskey*

A thesis submitted to the University of Huddersfield in partial  
fulfilment of the requirements for the degree of

*Doctor of Philosophy*

University of Huddersfield

School of Computing and Engineering

University of Huddersfield, Queensgate, Huddersfield, HD1 3DH, UK

December 2015



## Abstract

Due to the recent explosion of available data coming from the Web, sensor readings, social media, government authorities and scientific databases, both academia and industry have increased their interest in utilizing this knowledge. Processing huge amounts of data introduces several scientific and technological challenges, and creates new opportunities. Existing works on large-scale reasoning through mass parallelization (namely parallelization based on utilizing a large number of processing units) concentrated on monotonic reasoning, which can process only consistent datasets. The question arises whether and how mass parallelization can be applied to reasoning with huge amounts of imperfect (e.g. inconsistent, incomplete) information. Potential scenarios involving such imperfect data and knowledge include ontology evolution, ontology repair and smart city applications combining a variety of heterogeneous data sources. In this thesis, we overcome the limitations of monotonic reasoning, by studying several nonmonotonic logics that have the ability to handle imperfect knowledge, and it is shown that large-scale reasoning is indeed achievable for such complex knowledge structures. This work is mainly focused on adapting existing methods, thus ensuring that the proposed solutions are parallel and scalable. Initially, preliminaries and literature review are presented in order to introduce the reader to basic background and the state-of-the-art considering large-scale reasoning. Subsequently, each chapter presents an approach for large-scale reasoning over a given logic. Large-scale reasoning over defeasible logic is supported allowing conflict resolution by prioritizing the superiority among rules in the rule set. A solution for stratified semantics is presented where rules may contain both positive and negative subgoals, thus allowing reasoning over missing information in a given dataset. The approach for stratified semantics is generalized in order to fully support the well-founded semantics, where recursion through negation is allowed. Finally, conclusion includes observations from a preliminary investigation on a restricted form of answer set programming, a generic evaluation framework for large-scale reasoning, a discussion of the main findings of this work, and opportunities for future work.



## Acknowledgements

First of all, I would like to thank my supervisor Prof. Grigoris Antoniou, who guided me throughout these years. It was his outstanding supervision during my Master's degree back in Greece that convinced me to move to the UK and pursue my PhD studies at the University of Huddersfield. I am deeply grateful that he believed in my ideas, trusted me on solving various challenging topics and for his support in any arising issue.

I would like to thank my co-supervisor Prof. Thomas Leo McCluskey for his talks and advices. I would also like to thank PARK group members for all the interesting conversations that we had.

I would like to thank Prof. Wolfgang Faber and Dr Jeff Z. Pan for their participation in the committee and for their feedback that helped to improve this thesis.

I would like to acknowledge the financial support of the University of Huddersfield. I would like to thank the staff of the University of Huddersfield for helping me in several ways all these years.

Last, but not least, I would like to thank my family for their 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Research Questions and Contribution . . . . .	3
1.2	Thesis Structure . . . . .	4
1.3	Publications . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Preliminaries . . . . .	7
2.1.1	RDF/S . . . . .	7
2.1.2	OWL . . . . .	8
2.1.3	Description Logics . . . . .	8
2.1.4	Datalog . . . . .	9
2.1.5	Defeasible Logic . . . . .	10
2.1.6	Stratified Semantics . . . . .	13
2.1.7	Stratification - Defeasible Logic Versus Stratified Semantics . . . . .	14
2.1.8	Well-Founded Semantics . . . . .	15
2.1.9	Alternating Fixpoint Procedure . . . . .	17
2.1.10	Answer Set Programming . . . . .	18
2.2	Computing Models . . . . .	21
2.2.1	MapReduce Framework . . . . .	21
2.2.2	OpenMP . . . . .	23
2.2.3	MPI . . . . .	24
2.2.4	X10 . . . . .	26
<b>3</b>	<b>Related Work</b>	<b>27</b>
3.1	RDF/S Reasoning . . . . .	27
3.2	OWL Reasoning . . . . .	33
3.3	Description Logic Reasoning . . . . .	37
3.4	Conclusion . . . . .	41

<b>4</b>	<b>Defeasible Logic</b>	<b>43</b>
4.1	Single Variable Rule Sets . . . . .	43
4.2	Multi Variable Rule Sets . . . . .	46
4.2.1	Reasoning Overview . . . . .	47
4.2.2	Pass #1: Fired Rules Calculation . . . . .	49
4.2.3	Pass #2: Defeasible Reasoning . . . . .	52
4.2.4	Final Remarks . . . . .	54
4.3	Evaluation . . . . .	55
4.3.1	LUBM Use Case . . . . .	57
<b>5</b>	<b>Stratified Semantics of Logic Programs</b>	<b>61</b>
5.1	Algorithm Description . . . . .	61
5.1.1	Positive Goals Calculation . . . . .	63
5.1.2	Final Goal Calculation . . . . .	65
5.1.3	Special Cases . . . . .	66
5.1.4	Final Remarks . . . . .	67
5.2	Experimental Evaluation . . . . .	68
<b>6</b>	<b>Well-Founded Semantics</b>	<b>73</b>
6.1	Full Well-Founded Semantics . . . . .	73
6.1.1	Join and Anti-join for WFS . . . . .	73
6.1.2	Computing the Well-Founded Semantics . . . . .	77
6.1.3	Experimental Results . . . . .	83
6.2	Stratified Versus Full Well-Founded Semantics Approach . . . . .	87
6.2.1	Theoretical Analysis . . . . .	87
6.2.2	Experimental Analysis . . . . .	89
<b>7</b>	<b>Conclusion and Future Work</b>	<b>93</b>
7.1	Answer Set Programming . . . . .	93
7.1.1	Single Variable Programs . . . . .	93
7.1.2	Parallel Reasoning Based on Constants . . . . .	94
7.1.3	Parallel Reasoning Based on Predicates . . . . .	95
7.1.4	Final Remarks . . . . .	98
7.2	Evaluation Framework . . . . .	99
7.3	Discussion . . . . .	100
7.4	Future Work . . . . .	103



# List of Figures

2.1	Graphic representation of a triple. . . . .	8
2.2	Wordcount example. . . . .	22
4.1	Single variable inference . . . . .	44
4.2	Stratified rule set. Predicates are assigned to ranks. . . . .	46
4.3	Non-stratified rule set. Predicates cannot be assigned to ranks. . . . .	46
4.4	Overall reasoning process . . . . .	48
4.5	Fired rules calculation given the generic rule $r : A(X,Z), B(Z,Y) \{ \rightarrow, \Rightarrow, \rightsquigarrow \} [\neg] P(X,Y)$ . . . . .	50
4.6	Defeasible reasoning . . . . .	52
4.7	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). . . . .	58
4.8	Minimum, average and maximum reduce task runtime for each job with 40 reduce tasks. . . . .	58
4.9	Minimum, average and maximum reduce task runtime for each job with 400 reduce tasks. . . . .	59
5.1	Predicates assigned to ranks. . . . .	62
5.2	Runtime in minutes for join operations as a function of dataset size, for various numbers of nodes. . . . .	70
5.3	Runtime in minutes for anti-join operations as a function of dataset size, for various numbers of nodes. . . . .	70
5.4	Runtime in minutes for various numbers of rules and nodes. . . . .	71
5.5	Runtime in minutes for various matched values percentages. . . . .	71
6.1	<i>Win-not-win</i> test for cyclic datasets. Time in minutes as a function of dataset size, for various numbers of nodes. . . . .	85
6.2	<i>Win-not-win</i> test for tree-structured datasets. Time in minutes as a function of dataset size, for various numbers of nodes. . . . .	85

6.3	<i>Transitive closure with negation</i> test for chain-structured datasets. Time in minutes as a function of number of joins in the initially formed chain ( $\lceil n/k \rceil - 1$ ), for dataset size ( $n$ ) and number of facts per level ( $k$ ), comparing naive and optimized WFS fixpoint calculation. . . . .	86
6.4	<i>Transitive closure with negation</i> test for chain-structured datasets. Time in minutes as a function of dataset size ( $n$ ) and number of facts per level ( $k$ ) (constant $\lceil n/k \rceil - 1$ ), comparing naive and optimized WFS fixpoint calculation. . . . .	86
6.5	Time in minutes as a function of number of rules (ranks), comparing the stratified and the full WFS method. . . . .	90

# List of Tables

4.1	Rule set . . . . .	56
6.1	Speedup of the stratified over the full WFS method. . . . .	90



# Chapter 1

## Introduction

We are in the middle of the big data revolution: huge amounts of data are published by public and private organizations, and generated by sensor networks and social media. Apart from issues of size, this data is often dynamic and heterogeneous. In addition, data as a resource has been identified, and is increasingly utilized to generate added value; we are heading towards a data economy.

The challenge of big data is about managing it, but even more so about making sense of it: we want to avoid drowning in the sea of data, identify and focus on important aspects, and uncover hidden information and value. The role of data mining in this context is clear, but we believe reasoning has also an important role to play: for decision making, decision support, and for uncovering hidden knowledge in data, e.g. by deriving high-level information from low-level input data. Semantics has also an important role to play, both for understanding the data and for facilitating the combination of data from heterogeneous sources. The potential of semantics and reasoning in the context of big data has been well realized and is being debated<sup>1</sup>.

Big data poses great challenges to the reasoning and semantics communities, in terms of computational efficiency. The semantic web community has risen to this challenge through the use of mass parallelization (namely parallelization based on utilizing a large number of processing units) and approximation (which includes fuzzy reasoning and incomplete reasoning), and a lot has been achieved. Despite their importance in addressing large-scale reasoning, these works addressed forms of reasoning such as Datalog (query language used for deductive databases) and works on simple ontology (namely a specification of a conceptualization<sup>2</sup>) and RDFS reasoning (for details see Chapter 3), which can process only consistent datasets. In particular, all these works study monotonic forms of reasoning (namely reasoning where adding more knowledge to our knowledge base will not

---

<sup>1</sup>e.g. <http://lod2.eu/BlogPost/1698-eswc-2013-panel-semantic-technologies-for-big-data-analytics-opportunities-and-challenges.html>

<sup>2</sup><http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>

reduce the set of derived conclusions), and do not handle inconsistencies. Traditionally, the field of nonmonotonic reasoning (namely reasoning where adding more knowledge to our knowledge base may result in previously derivable conclusions to be no longer valid) addresses knowledge representation and reasoning issues related to incomplete and inconsistent information. Such imperfect information may naturally arise when information from independent sources is integrated. Indicative application scenarios of this kind of reasoning include:

- Decision making in smart environments [1].
- Rules with exceptions for decision making.
- Ontology diagnosis [2].
- Ontology evolution [3].
- Ontology repair [4].

From the practical point of view, nonmonotonic reasoning can provide the basis for various e-commerce applications, as discussed by Antoniou et al. [5], where the system acts as a broker, namely it matches the preferences of the buyer with the available offers provided by the seller. Antoniou et al. [5] presented a concrete example for apartment rental, where both buyer's preferences and available apartments specifications are encoded into defeasible logic, thus resolving arising conflicts and yielding suggestions whenever there is a match between demand and supply. In essence, this scenario can be extended to other commercial activities such as car purchase/rental and holiday packages.

In addition, there has been progress in the field of smart environments. More specifically, Nieves et al. [6] proposed a system that supports decision making in smart environments through argumentation. The architecture of the system is based on three rational agents, with each agent expressing its knowledge as extended logic programs. The aforementioned agents handle interactions within the smart environment, support and enhance both ongoing and predicted activities, and enhance the ability of human actors to perform certain activities. The Well-Founded Semantics is the underlying theory for the construction of arguments. As pointed out by Nieves et al. [6], their next steps will include the utilization of the approach for the Well-Founded Semantics, which is presented in this work, in order to scale their system up to big data.

Nonmonotonic approaches [7] and their adaptation to semantic web problems (e.g. Knorr et al. [8], Eiter et al. [9] and Antoniou et al. [10]) have been traditionally memory based. But this approach cannot be maintained in the face of big data. Especially, for the case of Linked Open Data<sup>3</sup>, with the Linked Open Data Cloud consisting of billions of triples, one could consider applying nonmonotonic reasoning in order to extract reacher information. This thesis aims at providing approaches that are able to overcome arising

---

<sup>3</sup><http://linkeddata.org/>

challenges that are related to the problem of large-scale reasoning with complex knowledge structures, in particular including inconsistencies and missing information.

## 1.1 Research Questions and Contribution

The main question of this research is: **How and to what extent large-scale reasoning with nonmonotonic and imperfect knowledge can be achieved through mass parallelization?** The answer to this question is the main contribution of this thesis, namely providing parallel and scalable approaches for nonmonotonic reasoning over various logics. More specifically, we consider nonmonotonic logics that can handle imperfect data by resolving conflicts (defeasible logic), handle incomplete data by performing reasoning over missing information (well-founded semantics), and handling both imperfect and missing information by supporting alternative world views (answer set programming).

**Can parallel defeasible reasoning be achieved for stratified (namely predicate dependencies are not allowed to form a cycle) and non-stratified (namely predicate dependencies that form a cycle are allowed) rule sets?** It is evident that parallel defeasible reasoning can be achieved for stratified rule sets by analyzing the dependencies within the given rule set and introducing a well-defined reasoning sequence (see Section 4.2.1). More specifically, the given rule set is divided into subsets where each subset (group of rules) is assigned a rank. At each rank, parallel defeasible reasoning is applied deriving new conclusions (see Sections 4.2.2 and 4.2.3), while reasoning is performed from lower to higher ranks. Full materialization is achieved once the whole rule set has been evaluated. On the other hand, for non-stratified rule sets scalable reasoning cannot be achieved by parallelizing a highly efficient serial algorithm introduced by Maher et al. [11] as the amount of the generated data is excessive for Big Data (see Section 4.2.4). We have detected the prime cause of limitation for non-stratified rule sets and several alternatives were considered. However, none of the alternatives could serve as a parallel and scalable approach that will retain the full semantics of the defeasible algorithm. Thus, a parallel approach for non-stratified rule sets that would produce sound and complete results is yet to be defined. For more details see Chapter 4.

**Can well-founded semantics be calculated by parallelizing inference?** Here we face the main problem of a three-valued logic, namely we need to avoid the computation of the entire Herbrand base so that the approach remains feasible for large-scale applications. Initially, we focused on stratified rule sets for which we defined a reasoning sequence that computes the closure (see Section 5.1). However, here we had the challenge of computing rules, in a parallel setting, that contained both positive and negative subgoals (see Sections 5.1 and 6.1.1). We provided a solution based on the assumption that each rule is safe, namely each variable in the head of the rule is also contained in a positive subgoal. Many proposed serial algorithms for the computation of the full well-founded semantics were based on calculating the entire Herbrand base, which was prohibiting for

Big Data. However, we managed to overcome this restriction by applying the *alternating fixpoint procedure* (see Sections 2.1.9 and 6.1). Thus, we have proposed both an approach that is tailored towards stratified rule sets and a generic approach for the full well-founded semantics. Note that although the approach for the full well-founded semantics is able to compute the closure of stratified rule sets as well, it is computationally more expensive compared to the proposed approach for stratified rule sets (see Section 6.2). For more details see Chapters 5 and 6.

**Which subsets of answer set programming can efficiently be parallelized in order to speed up the computation of answer sets?** We studied the applicability of parallel computation of answer set programs where all predicates are of arity one (see Section 7.1.1). Two different approaches are suggested: the first is based on parallelizing on constants (see Section 7.1.2), the second parallelizes on predicates (see Section 7.1.3). In each case, a method is provided that makes use of mass parallelization and calls standard ASP solvers as back-ends via an abstract API. For each method, we provided a theoretical analysis of its computational impact. Nevertheless, the presented approach cannot be extended as is for programs with predicates that have more than one argument. Such programs, in general, cannot be partitioned in independent segments, therefore requiring a more complex approach. Thus, we have provided a parallel and scalable approach for the computation of answer sets for monadic logic programs, while indicating the arising challenges for the general case (see Section 7.1.4). For more details see Section 7.1.

## 1.2 Thesis Structure

The thesis is organized as follows:

- **Chapter 2 provides an introduction to** various aspects of knowledge representation, basic notions of studied nonmonotonic logics, and computing models that can facilitate mass parallelization.
- **Chapter 3 presents a detailed literature review** for large-scale reasoning. In particular, scalable reasoning for RDF/S, Datalog, various OWL profiles (such as OWL Horst and OWL 2RL) and description logics. For each work, we provide a description of the approach and the main results of the experimental evaluation.
- **Chapter 4 presents a parallel and scalable approach for defeasible reasoning.** Initially, a restricted approach is considered namely defeasible reasoning for single variable rule sets. However, this approach is extended for rule sets that contain predicates of arbitrary arity (multi variable rule sets). Multi variable rule sets are divided into two categories, namely stratified and non-stratified rule sets. A parallel and scalable solution is presented for stratified rule sets, while for non-stratified rule sets we highlight the main arising challenges. This is an extended and



revised version of work that was published by Tachmazidis et al. [12, 13, 14].

- **Chapter 5 describes a solution for parallel and scalable reasoning over the stratified semantics of logic programs.** In particular, an efficient solution is presented for stratified rule sets, which is not applicable to non-stratified rule sets. However, this approach is computationally more efficient for stratified rule sets compared to the generic approach (see Chapter 6). This work has been included in Tachmazidis et al. [15].
- **Chapter 6 presents a solution for parallel and scalable reasoning over the full well-founded semantics.** More specifically, a generic approach, having the ability to compute the full well-founded semantics for any given rule set, is provided. This work has been included in Tachmazidis et al. [16, 17]. Note that in Tachmazidis et al. [17] algorithm description and experimental evaluation is based on X10 [18]. The approach over X10 can be found in Tachmazidis et al. [17], but it is not included in this thesis. Here, we provide both algorithm description and experiments based on the MapReduce framework. In addition, we show the advantages of the approach for stratified semantics (see Chapter 5) over the approach for full-well founded semantic for the subset of stratified programs.
- **Chapter 7 summarizes this work** by discussing observations from a preliminary investigation on single variable answer set programs, presenting a generic evaluation framework for large-scale reasoning, and providing a discussion of the main findings of this work and opportunities for future work.

### 1.3 Publications

This thesis either includes or is based on the work by Tachmazidis et al. [12, 13, 14, 15, 16, 17, 19] and Antoniou et al. [20].

- I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, “Towards parallel non-monotonic reasoning with billions of facts,” in KR, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, 2012.
- I. Tachmazidis, G. Antoniou, G. Flouris, S. Kotoulas, and L. McCluskey, “Large-scale Parallel Stratified Defeasible Reasoning,” in ECAI, ser. Frontiers in Artificial Intelligence and Applications, L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., vol. 242. IOS Press, 2012, pp. 738-743.
- I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, “Scalable Nonmonotonic Reasoning over RDF Data Using MapReduce,” in SSWS+HPCSW, 2012.

- G. Antoniou, J. Z. Pan, and I. Tachmazidis, “Large-scale complex reasoning with semantics: Approaches and challenges,” in *Web Information Systems Engineering - WISE 2013 Workshops - WISE 2013 International Workshops BigWebData, MBC, PCS, STeH, QUAT, SCEH, and STSC 2013*, Nanjing, China, October 13-15, 2013, Revised Selected Papers, ser. *Lecture Notes in Computer Science*, Z. Huang, C. Liu, J. He, and G. Huang, Eds., vol. 8182. Springer, 2013, pp. 1-10. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-54370-8\\_1](http://dx.doi.org/10.1007/978-3-642-54370-8_1)
- I. Tachmazidis and G. Antoniou, “Computing the Stratified Semantics of Logic Programs over Big Data through Mass Parallelization,” in *RuleML*, ser. *Lecture Notes in Computer Science*, L. Morgenstern, P. S. Stefaneas, F. Lévy, A. Wyner, and A. Paschke, Eds., vol. 8035. Springer, 2013, pp. 188-202.
- I. Tachmazidis, G. Antoniou, and W. Faber, “Efficient computation of the well-founded semantics over big data,” *TPLP*, vol. 14, no. 4-5, pp. 445-459, 2014. [Online]. Available: <http://dx.doi.org/10.1017/S1471068414000131>
- I. Tachmazidis, L. Cheng, S. Kotoulas, G. Antoniou, and T. E. Ward, “Massively parallel reasoning under the well-founded semantics using X10,” in *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, Limassol, Cyprus, November 10-12, 2014. IEEE Computer Society, 2014, pp. 162-169. [Online]. Available: <http://dx.doi.org/10.1109/ICTAI.2014.33>
- I. Tachmazidis, G. Antoniou, and W. Faber, “Computing Answer Sets for Monadic Logic Programs via Mapreduce,” in *ASPOCP*, 2014.

## Chapter 2

# Background

In this chapter, we provide a basic introduction to knowledge representation and describe the basic notions of several logics. For the nonmonotonic logics that have been studied in this work, we provide a more detailed description since certain details enable scalable and efficient reasoning. In addition, we present several well-known computing models that can facilitate mass parallelization.

### 2.1 Preliminaries

#### 2.1.1 RDF/S

RDF (Resource Description Framework) [5] is a data model where knowledge is represented in *subject-predicate-object* triples called *statements*. Such statements are used in order to represent existing entities (subject, object) and define the relations between these entities (predicate). In fact, RDF triples can be represented as a graph where *subject* and *object* are represented as nodes, while *predicate* is represented as a directed edge from *subject* to *object*. Figure 2.1 shows a graphic representation of the statement “This PhD thesis is written by Ilias Tachmazidis”.

In addition to data representation using RDF, we can define a vocabulary that would describe the domain we are working on, specifying the meaning of each class (subject, object) and property (predicate). Thus, one can use RDFS (RDF Schema) in order to provide an apt description of the modeled data, increasing inter-operability across domains. Using RDFS we can explicitly state the domain and range of property *writtenBy*, by setting class *Document* as domain and class *Student* as range. Such definitions allow us to assert whether “PhD thesis” is of type *Document* and whether “Ilias Tachmazidis” is of type *Student* in order to validate our model. For more details the reader is referred to [5].

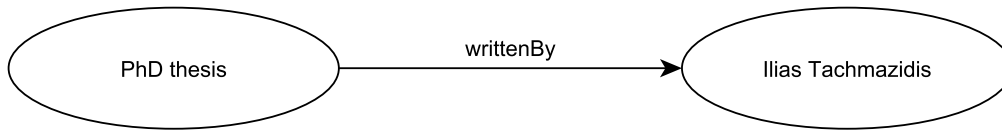


Figure 2.1: Graphic representation of a triple.

### 2.1.2 OWL

OWL (Web Ontology Language) [5] is a richer and more expressive language compared to RDF and RDFS. We can design a more descriptive domain model using OWL since we can define *disjointness* of classes, namely *students* are either *graduate* or *undergraduate*, but not both. In addition, classes can be defined as the *union*, *intersection* or *complement* of other classes. Using OWL, we can assert cardinality restrictions defining explicitly that each PhD student is assigned to two supervisors. Moreover, we can state that a property is *transitive* (e.g., a property such as *before*, namely if *A* is *before B* and *B* is *before C*, then *A* is also *before C*), or *inverse* of another property (e.g., the property *hasWritten* is *inverse* of property *writtenBy*).

There are three sublanguages of OWL, namely OWL Lite, OWL DL and OWL Full. As stated in [5]:

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

The basic intuition here is that OWL Lite is a subset of OWL DL, while OWL DL is a subset of OWL Full. In a nutshell, OWL Lite was designed to support classification hierarchy and simple constraints, OWL DL was designed to support maximum expressiveness possible without losing computational completeness and decidability of reasoning process, while OWL Full was designed to support maximum expressiveness with reasoning process being undecidable. For more details the reader is referred to [5].

### 2.1.3 Description Logics

Description Logics (DLs) [21] are a family of knowledge representation languages with expressivity between propositional logic and first-order logic (FOL). Description Logics are used in order to represent formally the underlying knowledge of a given domain. In particular, DLs model *concepts* (which correspond to: *unary predicates* in FOL and *classes* in OWL), roles (which correspond to: *binary predicates* in FOL and *properties* in OWL)

and individuals (which correspond to: *constants* in FOL and *individuals* in OWL). Roles and concepts are related with other roles and concepts through logical statements called *axioms*.

A DL knowledge base consists of two parts, namely the terminological part (TBox) and the assertional part (ABox). TBox defines the schema of a given DL, namely existing properties and relations between concepts and roles. Thus, a TBox statement would model knowledge such as “Every PhD student is a student”. On the other hand, ABox defines the facts of a given DL, namely properties of existing individuals. Thus, an ABox statement would model knowledge such as “Ilias Tachmazidis is a PhD student”. Considering the wide range of DLs, their naming defines also the underlying expressivity, with well-known medical informatics terminology bases, such as SNOMED-CT<sup>1</sup>, GALEN<sup>2</sup> and GO<sup>3</sup> being expressible in  $\mathcal{EL}$ .

#### 2.1.4 Datalog

Datalog is a declarative database query language that has been used extensively in the field of relational databases [22]. A Datalog program consists of function-free Horn rules that are applied on existing (explicit) facts in order to deduce new (implicit) facts from the given knowledge base. Considering the field of relational databases, facts can be perceived as rows in a table, with rules facilitating database queries, thus further enriching our knowledge base. Although Datalog was initially designed as a query language for deductive databases, in recent years, it has been widely utilized in various applications, such as declarative networking, program analysis, distributed social networking and security systems.

Given a datalog program and a knowledge base, there are mainly two approaches for deriving new knowledge, namely *top-down* and *bottom-up*. A *top-down* approach is designed to start with a given goal (or query), which is recursively reduced to subgoals, based on the body of each applicable rule, until the given subgoal corresponds to existing facts in our knowledge base. Thus, the formed resolution (inference) tree is built starting from the *top* goal and moving *down* to its subgoals. As opposed to *top-down*, *bottom-up* approach is based on the repetitive applications of the given rules on existing (explicit) and derived (implicit) facts until no new facts can be derived. Thus, once the full materialization of the knowledge base, based on the given rule set, is achieved then the facts that match the given query are used in order to provide the final answers. In essence, *top-down* approach is more query-oriented as it applies only rules that are relevant to the given query, on the cost of slower response compared to the *bottom-up* approach. On the other hand, *bottom-up* approach comes with a pre-computation overhead of materializing the entire knowledge base, on the benefit of quicker query-time responses compared to *top-down* as

---

<sup>1</sup><http://www.ihtsdo.org/snomed-ct>

<sup>2</sup><http://www.opengalen.org/>

<sup>3</sup><http://geneontology.org/>

all relevant (explicit and implicit) facts are part on the knowledge base.

### 2.1.5 Defeasible Logic

#### Defeasible Logic - Syntax

A defeasible theory [23, 7] (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

$$\begin{aligned} r &: \text{bird}(X) \Rightarrow \text{flies}(X) \\ r' &: \text{brokenWing}(X) \Rightarrow \neg \text{flies}(X) \end{aligned}$$

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.

### Defeasible Logic - Formal Definition

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

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

### Defeasible Logic - 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  $P = P(1), \dots, P(n)$  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  $P(i + 1) = +\Delta q$  if either  
 $q \in F$  or  
 $\exists r \in R_s[q] \forall \alpha \in A(r): +\Delta \alpha \in P(1..i)$

This means, 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 also be taken into account).

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

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 element  $\alpha$  of the antecedent for which we have established that  $\alpha$  is not definitely provable ( $-\Delta\alpha$ ).

$+\partial$ : We may append  $P(i + 1) = +\partial q$  if either

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

- (1)  $-\Delta q \in P(1..i)$  and
- (2) (2.1)  $\forall r \in R_{sd}[q] \exists \alpha \in A(r) : -\partial\alpha \in P(1..i)$  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  $t \not> s$

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  $\sim q$  is definitely provable (2.2); or there must be an applicable rule  $s$  with head  $\sim q$  such that no possibly applicable rule  $t$  with head  $q$  is superior to  $s$  (2.3).



### Defeasible Logic - Stratification

**Definition 2.1.1** *A rule set is stratified if all of its predicates can be assigned a rank such that*

- *no predicate depends on one of equal or greater rank*
- *no predicate is assigned a rank not equal to its complement*

*in any rule.*

#### 2.1.6 Stratified Semantics

In this work, we describe the *stratified semantics* of logic programming as they were defined by Gelder et al. [24] for stratified programs of the well-founded semantics. We impose several restrictions in order to achieve parallelization using the MapReduce framework.

**Definition 2.1.2** [24] *A general logic program is a finite set of general rules, which may have both positive and negative subgoals. A general rule is written with its head, or conclusion on the left, and its subgoal (body), if any to the right of the symbol “ $\leftarrow$ ”, which may be read “if”.  $\square$*

Given the following rule (see Gelder et al. [24]),

$$p(X) \leftarrow a(X), \mathbf{not} b(X).$$

$p(X)$  is the head,  $a(X)$  is a *positive subgoal*, and  $b(X)$  is a *negative subgoal*. This rule may be read as “ $p(X)$  if  $a(X)$  and not  $b(X)$ ”. A *Horn rule* is one with no negative subgoals, and a *Horn logic program* is one with only Horn rules.

We use the following conventions. A logical variable starts with a capital letter while a constant or a predicate starts with a lowercase letter. Note that functions are not allowed. A predicate of arbitrary arity will be referred as a *literal*. If  $p$  is a *positive literal* then  $\neg p$  is its *negative literal*,  $p$  and  $\neg p$  are *complements* of each other. Constants, variables and literals are *terms*. A *ground term* is a term with no variables. The *Herbrand universe* is the set of constants in a given program. The *Herbrand base* is the set of ground terms that are produced by the substitution of variables with constants in the Herbrand universe.

**Definition 2.1.3** [24] *A program is stratified if all of its predicates can be assigned a rank such that*

- *no predicate depends positively on one of greater rank, and*
- *no predicate depends negatively on one of equal or greater rank*

*in any rule.*  $\square$

**Definition 2.1.4** [24] *Given a program  $\mathbf{P}$ , a partial interpretation  $I$  is a consistent set of literals whose atoms are in the Herbrand base of  $\mathbf{P}$ . A total interpretation is a partial interpretation that contains every atom of the Hebrand base or its negation. We say a ground (variable-free) literal is true in  $I$  when it is in  $I$  and say it is false in  $I$  when its complement is in  $I$ . Similarly, we say a conjunction of ground literals is true in  $I$  if all of the literals are true in  $I$ , and is false in  $I$  if any of its literals is false in  $I$   $\square$*

According to the *stratified semantics* literals are classified as positive or negative as follows:

1. Facts are classified as positive literals.
2. All inferences for rank 0 are classified as positive literals, while those not inferred are classified as negative.
3. If all literals up to rank  $k - 1$  are classified either as positive or negative, then we can perform reasoning for rank  $k$ . All inferences for rank  $k$  are then classified as positive literals, while those not inferred are classified as negative.

**Definition 2.1.5** [24] *Let a program  $\mathbf{P}$ , its associated Herbrand base  $H$  and a partial interpretation  $I$  be given. We say  $A \subseteq H$  is an unfounded set (of  $\mathbf{P}$ ) with respect to  $I$  if each atom  $p \in A$  satisfies the following condition: For each instantiated rule  $R$  of  $\mathbf{P}$  whose head is  $p$ , (at least) one of the following holds:*

1. *Some (positive or negative) subgoal  $q$  of the body is false in  $I$ .*
2. *Some positive subgoal of the body occurs in  $A$ .*

*A literal that makes (1) or (2) above true is called a witness of unusability for rule  $R$  (with respect to  $I$ ).  $\square$*

### 2.1.7 Stratification - Defeasible Logic Versus Stratified Semantics

Note that the notion of stratification, in both Definition 2.1.1 and Definition 2.1.3, describes a way of assigning predicates to ranks such that there is no cycle among predicates that belong to different ranks. However, each definition is designed based on the semantics of the underlying logic and allows different structure of predicate dependencies. Here are the key differences between the two stratification definitions:

- For defeasible logic, every predicate that depends on another predicate is assigned a higher rank (see Definition 2.1.1). On the other hand, for stratified semantics, predicates are assigned a higher rank if they depend negatively on other predicates (see Definition 2.1.3).

- For defeasible logic, because every predicate that depends on another predicate is assigned a higher rank (see Definition 2.1.1), no cycles are allowed within a given rank. On the other hand, for stratified semantics, predicates that depend positively on other predicates can be assigned an equal rank (see Definition 2.1.3), and thus, are allowed to form positive cycles within a given rank.
- Defeasible logic contains both a literal  $p$  and its complement  $\neg p$ , which may require conflict resolution. Thus, both  $p$  and  $\neg p$  must be assigned to the same rank (see Definition 2.1.1). On the other hand, stratified semantics is based on negation as failure and do not handle conflict resolution using priorities over rules, and thus, do not need to include such conditions into the stratification definition (see Definition 2.1.3).
- Stratified semantics contain both positive and negative subgoals. Thus, a predicate that depends negatively on another predicate must be assigned a higher rank (see Definition 2.1.3) in order to allow reasoning over complete knowledge at each rank. On the other hand, defeasible logic does not define negative subgoals, and thus, do not need to include such conditions into the stratification definition (see Definition 2.1.1).

### 2.1.8 Well-Founded Semantics

In this subsection we provide the definition of the *well-founded semantics* (WFS) as they were defined by Gelder et al. [24].

**Definition 2.1.6** [24] *A general logic program is a finite set of general rules, which may have both positive and negative subgoals. A general rule is written with its head, or conclusion on the left, and its subgoal (body), if any to the right of the symbol “ $\leftarrow$ ”, which may be read “if”.*

Given the following rule (see Gelder et al. [24]),

$$p(X) \leftarrow a(X), \mathbf{not} \ b(X).$$

$p(X)$  is the head,  $a(X)$  is a *positive subgoal*, and  $b(X)$  is a *negative subgoal*. This rule may be read as “ $p(X)$  if  $a(X)$  and not  $b(X)$ ”. A *Horn rule* is one with no negative subgoals, and a *Horn logic program* is one with only Horn rules.

We use the following conventions. A logical variable starts with a capital letter while a constant or a predicate starts with a lowercase letter. Note that functions are not allowed. A predicate of arbitrary arity will be referred as a *literal*. Constants, variables and literals are *terms*. A *ground term* is a term with no variables. The *Herbrand universe* is the set of constants in a given program. The *Herbrand base* is the set of ground terms that are produced by the substitution of variables with constants in the Herbrand universe. We will refer to Horn rules also as *definite* rules, likewise Horn programs will also be referred to as *definite* programs.

**Definition 2.1.7** [24] *The Herbrand instantiation of a general logic program is the set of rules obtained by substituting terms in the Herbrand universe for variables in every possible way. An instantiated rule is one in the Herbrand instantiation. Whereas “uninstantiated” logic programs are assumed to be a finite set of rules, instantiated logic programs may well be infinite.*

Note that this definition is a quote from [24]. The last statement does not apply to the setting in this work: instantiated programs are always finite.

**Definition 2.1.8** [24] *Given a program  $\mathbf{P}$ , a partial interpretation  $I$  is a consistent set of literals whose atoms are in the Herbrand base of  $\mathbf{P}$ . A total interpretation is a partial interpretation that contains every atom of the Herbrand base or its negation. We say a ground (variable-free) literal is true in  $I$  when it is in  $I$  and say it is false in  $I$  when its complement is in  $I$ . Similarly, we say a conjunction of ground literals is true in  $I$  if all of the literals are true in  $I$ , and is false in  $I$  if any of its literals is false in  $I$ .*

**Definition 2.1.9** [24] *Let a program  $\mathbf{P}$ , its associated Herbrand base  $\mathbf{H}$  and a partial interpretation  $I$  be given. We say  $A \subseteq \mathbf{H}$  is an unfounded set (of  $\mathbf{P}$ ) with respect to  $I$  if each atom  $p \in A$  satisfies the following condition: For each instantiated rule  $R$  of  $\mathbf{P}$  whose head is  $p$ , (at least) one of the following holds:*

1. *Some (positive or negative) subgoal  $q$  of the body is false in  $I$ .*
2. *Some positive subgoal of the body occurs in  $A$ .*

*A literal that makes (1) or (2) above true is called a witness of unusability for rule  $R$  (with respect to  $I$ ).*

**Theorem 2.1.1** [24] *The data complexity of the well-founded semantics for function-free programs is polynomial time.*

In this work, we require each rule to be safe, that is, each variable in a rule must occur (also) in a positive subgoal. Safe programs consist of safe rules only. This safety criterion is an adaptation of range restriction [25], which guarantees the important concept of domain independence, originally studied in deductive databases (see for example Abiteboul et al. [26]). Domain independence means that the outcome of a query does not depend on the domain. In our setting, this means that the well-founded model will stay the same, no matter whether the Herbrand universe or any superset of it is considered. Therefore, the computed results of a safe program are robust with respect to extensions of the rule or fact base. Especially irrelevant additions of new constants will have no effect on computed models, a property which cannot be guaranteed for unsafe programs. Apart from this semantic property, the safety condition implicitly also enforces a certain locality of computation, which is important for our proposed method, as we shall discuss in subsection 6.1.2.

### 2.1.9 Alternating Fixpoint Procedure

In this subsection, we provide the definition of the alternating fixpoint procedure as it was defined by Brass et al. [27].

**Definition 2.1.10** [27] *For a set  $S$  of literals we define the following sets:*

$$\begin{aligned} \text{pos}(S) &:= \{A \in S \mid A \text{ is a positive literal}\}, \\ \text{neg}(S) &:= \{A \mid \mathbf{not} A \in S\}. \end{aligned}$$

**Definition 2.1.11** [27] (Extended Immediate Consequence Operator)

*Let  $P$  be a normal logic program. Let  $I$  and  $J$  be sets of ground atoms. The set  $T_{P,J}(I)$  of immediate consequences of  $I$  w.r.t.  $P$  and  $J$  is defined as follows:*

$$T_{P,J}(I) := \{A \mid \text{there is } A \leftarrow \mathcal{B} \in \text{ground}(P) \text{ with } \text{pos}(\mathcal{B}) \subseteq I \text{ and } \text{neg}(\mathcal{B}) \cap J = \emptyset\}.$$

*If  $P$  is definite, the set  $J$  is not needed and we obtain the standard immediate consequence operator  $T_P$  by  $T_P(I) = T_{P,\emptyset}(I)$ .*

For an operator  $T$  we define  $T \uparrow 0 := \emptyset$  and  $T \uparrow i := T(T \uparrow i - 1)$ , for  $i > 0$ .  $\text{lfp}(T)$  denotes the least fixpoint of  $T$ , i.e. the smallest set  $S$  such that  $T(S) = S$ .

**Definition 2.1.12** [27] (Alternating Fixpoint Procedure)

*Let  $P$  be a normal logic program. Let  $P^+$  denote the subprogram consisting of the definite rules of  $P$ . Then the sequence  $(K_i, U_i)_{i \geq 0}$  with set  $K_i$  of true (known) facts and  $U_i$  of possible (unknown) facts is defined by:*

$$\begin{aligned} K_0 &:= \text{lfp}(T_{P^+}) & U_0 &:= \text{lfp}(T_{P,K_0}) \\ i > 0 : & K_i &:= \text{lfp}(T_{P,U_{i-1}}) & U_i &:= \text{lfp}(T_{P,K_i}) \end{aligned}$$

*The computation terminates when the sequence becomes stationary, i.e., when a fixpoint is reached in the sense that*

$$(K_i, U_i) = (K_{i+1}, U_{i+1}).$$

*This computation schema is called the Alternating Fixpoint Procedure (AFP).*

We rely on the definition of the *well-founded partial model*  $W_p^*$  of  $P$  as given by Gelder et al. [24].

**Theorem 2.1.2** [27] (Correctness of AFP)

*Let the sequence  $(K_i, U_i)_{i \geq 0}$  be defined as above. Then there is a  $j \geq 0$  such that  $(K_j, U_j) = (K_{j+1}, U_{j+1})$ . The well-founded model  $W_p^*$  of  $P$  can be directly derived from the fixpoint  $(K_j, U_j)$ , i.e.,*

$$W_p^* = \{L \mid \begin{array}{l} L \text{ is a positive ground literal and } L \in K_j \text{ or} \\ L \text{ is a negative ground literal} \\ \text{not } A \text{ and } A \in \text{BASE}(P) - U_j \}, \end{array}$$

where  $\text{BASE}(P)$  is the Herbrand base of program  $P$ .

**Lemma 2.1.1** [27] (Monotonicity)

Let the sequence  $(K_i, U_i)_{i \geq 0}$  be defined as above. Then the following holds for  $i \geq 0$ :  
 $K_i \subseteq K_{i+1}$ ,  $U_i \supseteq U_{i+1}$ ,  $K_i \subseteq U_i$ .

### 2.1.10 Answer Set Programming

Here we provide the basic notions of answer set programming (ASP) as they were defined by Baral [28] and Lifschitz [29].

**Definition 2.1.13** [28] *The axiom alphabet (or simply the alphabet) of an answer set framework consists of seven classes of symbols:*

1. variables,
2. object constants (also referred to as constants),
3. function symbols,
4. predicate symbols,
5. connectives,
6. punctuation symbols, and
7. the special symbol  $\perp$ ;

where the connectives and punctuation symbols are fixed to the set  $\{\neg, \text{or}, \leftarrow, \text{not}, ', '\}$  and  $\{ '(', ')', ', ' \}$  respectively; while the other classes vary from alphabet to alphabet.

**Definition 2.1.14** [28] *A term is inductively defined as follows:*

1. A variable is a term.
2. A constant is a term.
3. If  $f$  is an  $n$ -ary function symbol and  $t_1, \dots, t_n$  are terms then  $f(t_1, \dots, t_n)$  is a term.

**Definition 2.1.15** [28] *A term is said to be ground, if no variable occurs in it.*

**Definition 2.1.16** [28] *Herbrand Universe and Herbrand Base*

- The Herbrand Universe of a language  $\mathcal{L}$ , denoted by  $\text{HU}_{\mathcal{L}}$ , is the set of all ground terms which can be formed with the functions and constants in  $\mathcal{L}$ .

- An atom is of the form  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate symbol and each  $t_i$  is a term. If each of the  $t_i$ s is ground then the atom is said to be ground.
- The Hebrant Base of a language  $\mathcal{L}$ , denoted by  $HB_{\mathcal{L}}$ , is the set of all ground atoms that can be formed with predicates from  $\mathcal{L}$  and terms from  $HU_{\mathcal{L}}$ .
- A literal is either an atom or an atom preceded by the symbol  $\neg$ . The former is referred to as a positive literal, while the latter is referred to as a negative literal. A literal is referred to as ground if the atom in it is ground.
- A naf-literal is either an atom or an atom preceded by the symbol **not**. The former is referred to as a positive naf-literal, while the later is referred to as a negative naf-literal.
- A gen-literal is either a literal of a literal preceded by the symbol **not**.

**Definition 2.1.17** [28] A rule is of the form:

$$L_0 \text{ or } \dots \text{ or } L_k \leftarrow L_{k+1}, \dots, L_m, \text{ not } L_{m+1}, \dots, \text{ not } L_n$$

where  $L_i$ s are literals or when  $k = 0$ ,  $L_0$  may be the symbol  $\perp$ , and  $k \geq 0$ ,  $m \geq k$ , and  $n \geq m$ .

A rule is said to be ground if all the literals of the rule are ground.

The parts on the left and on the right of ' $\leftarrow$ ' are called the head (or conclusion) and the body (or premise) of the rule, respectively.

A rule with an empty body and a single disjunct in the head (i.e.,  $k=0$ ) is called a fact, and then if  $L_0$  is a ground literal we refer to it as a ground fact.

A fact can be simply written without the  $\leftarrow$  as:

$$L_0.$$

When  $k = 0$ , and  $L_0 = \perp$ , we refer to the rule as a constraint.

The  $\perp$ s in the heads of constraints are often eliminated and simply written as rules with empty head, as in

$$\leftarrow L_1, \dots, L_m, \text{ not } L_{m+1}, \dots, \text{ not } L_n$$

**Definition 2.1.18** [28] Let  $r$  be a rule in a language  $\mathcal{L}$ . The grounding of  $r$  in  $\mathcal{L}$ , denoted by  $\text{ground}(r, \mathcal{L})$ , is the set of all rules obtained from  $r$  by all possible substitutions of elements of  $HU_{\mathcal{L}}$  for the variables in  $r$ .

**Definition 2.1.19** [28] The answer set language given by an alphabet consists of the set of all ground rules constructed from the symbols of the alphabet.

In addition to the above definitions, in this work we use the following conventions. A variable starts with a capital letter, a constant or a predicate starts with a lowercase letter, while functions are not allowed. We focus on *monadic* or *single-variable* literals, namely literals containing only one term. A set of literals  $X$  is said to be *consistent* if, for every literal  $p \in X$ , its complementary literal is not contained in  $X$ .

In this work, we focus on *monadic* or *single-variable* rules, namely rules of the following form,

$$p_n(Y) \leftarrow a_1(Y), \dots, a_k(Y), \text{not } b_1(Y), \dots, \text{not } b_m(Y). \quad (2.1)$$

Programs containing only *monadic* or *single-variable* rules will be called *monadic* or *single-variable* programs. It is evident from the definition above that all allowed rules are safe, namely each variable in a rule occurs (also) in a positive subgoal. In addition all allowed programs are safe as they consist of safe rules only.

Note that in (2.1) we used in each literal the variable  $Y$  (as opposed to traditionally used variable  $X$ ) in order to avoid any confusion with the set of literals  $X$ . However, the name of the used variable does not affect the definition of a rule. Thus, the following two rules

$$\begin{aligned} p_n(Y) &\leftarrow a_1(Y), \dots, a_k(Y), \text{not } b_1(Y), \dots, \text{not } b_m(Y). \\ p_n(Z) &\leftarrow a_1(Z), \dots, a_k(Z), \text{not } b_1(Z), \dots, \text{not } b_m(Z). \end{aligned}$$

are considered equivalent.

We use the definition of answer sets as it was defined by Lifschitz [29] with minor changes for consistency and readability purposes, taking into consideration the conventions above.

**Definition 2.1.20** [29] *The notion of an answer set is defined first for programs that do not contain negation as failure ( $m = 0$  in every (instantiated) rule (2.1) of the program). Let  $\Pi$  be such a program, and let  $X$  be a consistent set of literals. We say that  $X$  is closed under  $\Pi$  if, for every rule in  $\Pi$ ,  $\text{Head} \cap X \neq \emptyset$  whenever  $\text{Body} \subseteq X$ . We say that  $X$  is an answer set for  $\Pi$  if  $X$  is minimal among the sets closed under  $\Pi$  (relative to set inclusion).*

*For instance, the program*

$$\begin{aligned} &p; q, \\ \neg r &\leftarrow p \end{aligned} \quad (2.2)$$

*has two answer sets:*

$$\{p, \neg r\}, \{q\}, \quad (2.3)$$

*If we add the constraint*

$$\leftarrow q \quad (2.4)$$



to (2.2), we will get a program whose only answer set is the first of sets (2.3). On the other hand, if we add the rule

$$\neg q$$

to (2.2), we will get a program whose only answer set is  $\{p, \neg q, \neg r\}$ .

To extend the definition of an answer set to programs with negation as failure, take an arbitrary program  $\Pi$ , and let  $X$  be a consistent set of literals. The reduct  $\Pi^X$  of  $\Pi$  relative to  $X$  is the set of rules

$$p_n(\text{const}_i) \leftarrow a_1(\text{const}_i), \dots, a_k(\text{const}_i).$$

for all (instantiated) rules (2.1) in  $\Pi$  such that  $X$  does not contain any of  $b_1(\text{const}_i), \dots, b_m(\text{const}_i)$  (where  $\text{const}_i$  represents every constant in the Herbrand universe). Thus  $\Pi^X$  is a program without negation as failure. We say that  $X$  is an answer set for  $\Pi$  if  $X$  is an answer set for  $\Pi^X$ .

## 2.2 Computing Models

### 2.2.1 MapReduce Framework

MapReduce is a framework for parallel processing over huge datasets [30]. 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.

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 the following documents as input:

Doc1: "Hello world."  
 Doc2: "Hello MapReduce."

Figure 2.2 depicts the whole process. During map phase, each map operation gets as input a line of a document. *Map* function extracts words from each line and emits pairs of the form  $\langle w, "1" \rangle$  meaning that word  $w$  occurred once ("1"), namely the following pairs:

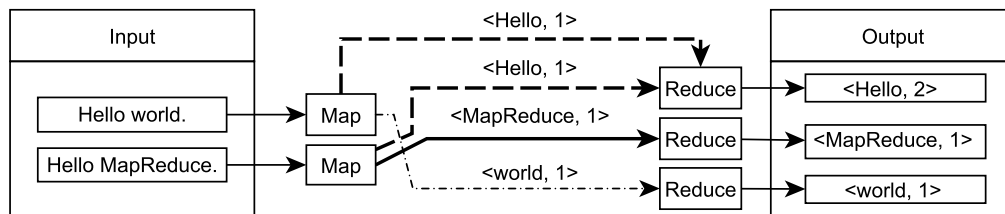


Figure 2.2: Wordcount example.

**Algorithm 1** Wordcount example

---

```

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

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

```

---

```

<Hello, 1>
<world, 1>
<Hello, 1>
<MapReduce, 1>

```

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

```

<Hello, <1,1>>
<MapReduce, 1>
<world, 1>

```

During the reduce phase, 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. Thus, the reducer with key:

```

Hello will emit <Hello, 2>
MapReduce will emit <MapReduce, 1>
world will emit <world, 1>

```

Here is a list of MapReduce advantages:

- Can process large amounts of input/output data.
- Developer writes few routines which are following the general interface.
- Provides automated management of fault tolerance.

Here is a list of MapReduce disadvantages:

- MapReduce interface, while generic, does not necessarily fit to any given problem.
- Not well-suited for applications that require multiple iterations over data where only a small subset is changing.
- Relies heavily on HDFS, which results in heavy I/O overheads.

### 2.2.2 OpenMP

OpenMP<sup>4</sup> (Open Multi-Processing) is an Application Program Interface (API) that has been designed in order to facilitate the usage of multi-threaded (concurrent threads within a process) and shared memory parallelism (parallelism over a single machine where all threads have access to a shared global memory). OpenMP supports programming in C, C++ and Fortran, and it has been designed to operate over a variety of platforms. The proposed API consists of mainly three components: (a) Compiler Directives, (b) Runtime Library Routines, and (c) Environment Variables. *Compiler directives* can be used for defining a parallel region, parallelizing blocks of code or loop iterations through the use of threads, and synchronizing thread execution. *Runtime library routines* can be used for managing various details of threads, and handling parallelism, parallel regions and locks. *Environment Variables* can be used during run-time for defining the number of threads and loop iterations behavior, and managing threads and the level of nested parallelism.

Algorithm 2<sup>5</sup> shows how OpenMP can be combined with C/C++ in order to facilitate parallel execution. Here, we use a compiler directive (`#pragma omp parallel private(tid)`) in order to indicate that the given block of code will be executed in parallel. In addition, runtime library routines are used in order to retrieve the id of the thread that is running the code (`omp_get_thread_num()`) and output a “hello world” message, while for the master thread we also retrieve (`omp_get_num_threads()`) and output the total number of threads. An indicative output, using 4 threads, could be the following:

```
Hello World from thread = 0
Hello World from thread = 3
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 1
```

---

<sup>4</sup><http://openmp.org/wp/>

<sup>5</sup><https://computing.llnl.gov/tutorials/openMP/>

---

**Algorithm 2** OpenMP example in C/C++

---

```

#include <omp.h>
main () {
    int nthreads, tid;
    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
}

```

---

Here is a list of OpenMP advantages:

- Incremental parallelism is supported, namely the programmer can parallelize gradually different parts of the program.
- Produced code works both in serial and parallel without further adjustments.
- Data decomposition is handled automatically.

Here is a list of OpenMP disadvantages:

- Can only utilize shared memory parallelism.
- Scalability is limited by memory architecture.
- Developer should be aware of synchronization bugs and race conditions.

### 2.2.3 MPI

MPI (Message Passing Interface) is a specification for the developers and users of message passing libraries<sup>6</sup>. Thus, MPI provides the specifications for any given library while not being a library itself. Essentially, it describes how various processes can communicate with each other using a predefined set of operations. Although MPI was initially intended for distributed memory architectures, it gradually incorporated shared memory settings as well, operating nowadays on either or both distributed and shared memory architectures.

---

<sup>6</sup><https://computing.llnl.gov/tutorials/mpi/>

---

**Algorithm 3** MPI example in C/C++

---

```

#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv)
{
    int ierr, num_procs, my_id;
    ierr = MPI_Init(&argc, &argv);
    /* find out MY process ID, and how many processes were started. */
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    printf("Hello world! I'm process %i out of %i processes\n", my_id, num_procs);
    ierr = MPI_Finalize();
}

```

---

Various incentives drove the community to the development of MPI. Amongst others, MPI provided standardization, namely it could be supported easier across platforms, thus reducing the cost of code implementation.

Algorithm 3<sup>7</sup> presents a program written in C that is using MPI. First, we need to initialize the MPI execution environment (`MPI_Init(&argc, &argv)`), subsequently get the unique id of the process (`MPI_Comm_rank(MPI_COMM_WORLD, &my_id)`) and the total number of MPI processes (`MPI_Comm_size(MPI_COMM_WORLD, &num_procs)`), output a “hello world” message, and finally terminate the MPI execution environment (`MPI_Finalize()`). An indicative output, using 4 processes, could be the following:

```

Hello world! I'm process 3 out of 4 processes.
Hello world! I'm process 0 out of 4 processes.
Hello world! I'm process 2 out of 4 processes.
Hello world! I'm process 1 out of 4 processes.

```

Here is a list of MPI advantages:

- Runs on distributed and shared memory platforms.
- Applies to a wide range of problems by providing a certain level of abstraction.
- Allows for (hardware) cost management.

Here is a list of MPI disadvantages:

- Serial to parallel implementation is not easy to achieve.
- Hard to debug due to the high level of parallelization.
- Communication overhead may become a bottleneck.

---

<sup>7</sup><http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml>

### 2.2.4 X10

X10 [18] is a new multi-paradigm programming language developed by IBM. It supports the asynchronous partitioned global address space (APGAS) model and is specifically designed to increase programmer productivity, while being amenable to programming shared memory and distributed memory supercomputers. It uses the concepts of `place` and `activity` as the kernel notions to exploit parallelism in the available hardware. A `place` is a logical abstraction of the underlying heterogeneous processing element in the hardware, such as cores in a multi-core architecture, GPUs, or an entire physical machine. `Activities` are light-weight threads that run on `places`. X10 schedules activities on `places` to best utilize the available parallelism. The number of `places` is constant through the lifetime of an X10 program and is initialized at program startup. `Activities` on the other hand can be forked at program execution time. Forking an activity can be blocking, wherein the parent returns after the forked activity completes execution, or non-blocking, wherein the parent returns instantaneously, after forking an activity. Furthermore, these activities can be forked locally or on a remote `place`.

X10 provides a data structure called distributed array (`DistArray`) for programming parallel algorithms. One or more elements in the `DistArray` can be mapped to a single `place` using the concept of `points` [18], and such elements can be kept in or operated on memory through the life of the code. The following three X10 primitives are critical in understanding the pseudocode given in the following sections:

- `at(p) S`: this construct executes statement `S` at a specific `place p`. The current activity is blocked until `S` finishes executing on `p`.
- `async S`: a child activity is forked by this construct. The current activity returns immediately (non-blocking) after forking `S`.
- `finish S`: this construct is used to block the current activity and wait for all activities forked by `S` to terminate.

Here is a list of X10 advantages:

- Flexible and efficient scheduling where tasks are separated from the underlying concurrency model, thereby allowing one to implement an efficient scheduling strategy.
- Can utilize both shared and distributed memory settings.
- Allows implementations that fit to various problems.

Here is a list of X10 disadvantages:

- Developer needs to handle manually operations such as data and task distribution.
- Memory capacity of the used architecture may limit scalability.
- Requires experienced developers in order to use efficiently the provided flexibility.

# Chapter 3

## Related Work

In this chapter, we present the related work on large-scale reasoning. In particular, we focus on parallel and scalable solutions that enable large-scale reasoning for various logics. We classify related work in the following three categories: (a) RDF/S (b) OWL, and (c) Description Logics. Although there is a clear overlap between OWL and Description Logics, we consider them separately by classifying works based on well-known OWL rule sets (such as OWL Horst and OWL 2RL) to OWL, and works covering an arbitrary expressiveness of a given Description Logic (such as  $\mathcal{ALC}$  and  $\mathcal{SHN}$ ) to Description Logics. In addition, when a given work covers more than one logic (e.g., both RDFS and OWL Horst), then this work is classified to the most generic logic. To the best of our knowledge, there is no existing work in the literature for large-scale nonmonotonic reasoning.

### 3.1 RDF/S Reasoning

**Logic: RDFS.** MARVIN (MAssive RDF Versatile Inference Network) [31] (see also MARVIN [32]) is a parallel and distributed platform for massive processing of RDF data based on a peer-to-peer model. It implements the *divide-conquer-swap* strategy, which partitions the given dataset into subsets (divide), computes the closure (conquer) and repartitions data by exchanging triples among neighboring nodes (swap). Closure is computed by utilizing reasoners as an external library, with eventually completeness of inference guaranteed, since triples that may emit new knowledge are gradually collocated on the same node. In addition, in order to minimize memory and bandwidth overheads the system supports duplicate detection and removal.

MARVIN achieves a balance between random and deterministic approach, combining load balancing with efficient derivation. More specifically, the random approach (triples are exchanged randomly among nodes) provides good load balancing properties, but is highly inefficient since triples appear on random nodes, thus postponing derivation. On the contrary, the deterministic approach (triples are redirected to a specific node according

to their key) is efficient as triples that will lead to derivations are located on the same node, but at the cost of highly unbalanced workloads for skewed dataset distributions. The implemented SpeedDate algorithm swaps triples within a neighborhood of nodes, thus popular keys are distributed over several nodes. Triples are constantly located within a restricted subsets of nodes providing a certain degree of determinism.

Authors studied various metrics by providing simulated results. Simulations deal with the number of nodes, number of items in the system, various data distributions, nodes availability during the reasoning process, recall, load balancing and scalability. The system surpasses the random approach in terms of efficiency, showing similar load balancing properties, and the deterministic approach in terms of scalability, with the ability to utilize higher degree of parallelization. Considering scalability, the average throughput per node follows a square root curve with respect to the number of items. In addition, authors provide experiments on real RDF data. The system is shown to scale to up to 64 nodes processing datasets of up to 14.9M triples. MARVIN shows sublinear speedups while performing better when low tolerance to duplicates is allowed.

**Logic: RDFS.** Kotoulas et al. [33] point out the fact that Semantic Web data follows highly skewed term distribution. Such term distributions pose a significant challenge for parallel reasoning since achieving high degree of parallelization requires balanced workloads coming from even data distribution among nodes. This becomes a scalability issue for term-based partitioning, an approach where triples that share common terms are collocated on the same node in order to be matched and derive new conclusions. By analyzing several real-world datasets, it is evident that in some datasets the most popular term can be found in up to 55% of all triples in dataset. The most popular term will be redirected to a single node that will be responsible for handling all corresponding triples. Thus, in case the most popular term appears in 10%-20% of triples in the given dataset then only a speedup of 5-10x can be obtained, while adding more nodes will not provide any further speedup.

The scalability of the term-based partitioning is limited by the frequency of the most popular term in the dataset. On the other hand, triples can be distributed randomly, providing balanced workload and high scalability at the cost of a highly inefficient derivation process. Thus, an approach called *speeddate* is proposed, where triples are distributed based on their terms over a neighborhood of nodes, providing both the efficiency of term-based partitioning and the scalability of the random distribution. Speeddate is based on distributed clustering, where each node decides deterministically which triples to exchange with its neighbors. Thus, neighborhoods (regions) emerge based on the terms in the given dataset requiring no upfront data analysis. In addition, such regions are elastic in the sense that they are larger for popular terms, thus providing load balancing, while no central coordination is required for handling the network.

Experimental evaluation of speeddate was performed on MARVIN [31] showing the scalability of the proposed approach. Various evaluation criteria were considered such



as *joins*, *produced triples*, *transferred data* and *number of nodes*. Data clustering was considered, with *random* clustering providing the lowest join throughput (number of joins per seconds). On the other hand, the *full* clustering approach that combined both *local* clustering (triples with same terms are loaded from local storage for joins) and *distributed* clustering (following the speeddate algorithm) provided the highest join throughput. In addition, a variation of full clustering called *compound* clustering was tested, where keys comprised of various combinations of terms, decreasing however join throughput. The system showed good scalability properties for up to 64 nodes for inputs of up to 195M triples. Finally, authors ensured that the system provides reasoning completeness, while reaching load-balancing properties of an uniform random partitioning.

**Logic: RDFS.** Goodman et al. [34] present a parallel approach for dictionary encoding, RDFS inference and SPARQL query processing on the Cray XMT supercomputer. Cray XMT is designed based on shared-memory architecture in order to absorb memory and network latency, while providing high level of parallelization. All processes are performed completely in-memory by utilizing the global shared-memory. In particular, authors present an algorithm for encoding RDFS triples into a set of 64-bit integers, thus providing a mapping between string and integer representation. It is required that RDFS triples are represented in either N-Triples or N-Quads format. Dictionary encoding is highly optimized for parallel processing while allowing updates to an existing dictionary. Final representation consists of triples encoded as integer values, and the mappings from each unique string to its corresponding integer value.

The aforementioned in-memory representation is consequently utilized for RDFS closure computation. RDFS closure is based on a previously presented algorithm [35]. However, the approach is altered in order to reduce memory usage, with the new approach optimizing the use of hash key values in order to include information about the availability of each slot (where all free slots are assigned a predefined key value). In addition, RDFS reasoning is based on processing the entire dataset instead of using queues that contain matching triples for each rule. However, such memory optimization resulted in increased computation time. Subsequently, RDFS closure is transformed into a graph representation in order to facilitate SPARQL query answering. An algorithm called Sprinkle SPARQL is used in order to estimate the set of matching possibilities, while query answering gradually identifies matching triples and calculates final results by combining all intermediate results.

Experimental evaluation showed that the system is able to scale up to 512 processors while handling 20 billion triples completely in-memory using the Cray XMT supercomputer. In particular, dictionary encoding was tested on datasets of up to 16.5 billion triples, coming with compression ratio ranging from 3.2 to 4.4, and speedups ranging from 2.4 to 3.3. RDFS closure generated 20.1 billion unique triples, requiring 40% less memory at the cost of 11% to 33% increase in computation time, with speedups ranging from 6 to 9, where. SPARQL queries showed speedups ranging from 4.3 to 28 for complicated

queries. Nevertheless, authors point out the fact that Sprinkle SPARQL do not perform well for simple queries as it comes with a significant overhead. Thus, Sprinkle SPARQL did not manage outperform all evaluated alternatives for simple queries. It is also shown throughout the evaluation of each operation that for increasing number of used processors, the required time decreases.

**Logic: RDFS.** Weaver et al. [36] consider parallel materialization of RDFS closure using the Message Passing Interface (MPI). In order to justify the correctness of the proposed parallel algorithm, authors provide a theoretical analysis of their approach. A special subset of rules is identified, namely rules that match only one assertional (instance) triple, while all other antecedents are matched by ontological (schema) triples. Such rules allow for efficient data partitioning where instance triples are divided into several partitions while schema triples are copied to each partition. Rules are applied at each partition in parallel. In addition, new conclusions are added to the partition from which they were derived in order to allow further derivations. The proposed reasoning process is proved to be sound and complete since none of the RDFS rules violates the premises for the partitioning. The algorithm for parallel RDFS inferencing is provided, showing that the union of all derivations constitutes the finite RDFS closure.

Data are partitioned evenly across processes requiring no preprocessing. Thus, each process receives a subset of the given dataset while partitioning is facilitated by the input format, namely triples follow the N-triples format (where each line contains a triple in a “subject predicate object” representation). Blank nodes pose a challenge in a distributed setting as they are unique only within a given graph. However, a solution is provided by encoding already existing blank nodes into URIs, while for rules that generate blank nodes allocated to literals, a URI is generated by encoding the literal. The implementation is written in C where MPI is used for parallel I/O and interprocess communication (when reading instance triples). Rule applications are performed using C libraries for in-memory RDF storage and query processing. Thus, once triples are loaded into memory, each rule is expressed and applied as a SPARQL query. Each partition derives its own conclusions with no support for duplicate elimination across partitions as each process outputs its derivations to a separate file.

Experimental evaluation showed the performance of the system for up to 128 processes with the ability to handle inputs of up to 345M triples. The performance of inferencing scales linearly for increasing number of triples and number of processes. However, by taking into consideration overall runtimes, it is evident that, for small datasets, the speedup decreases for larger numbers of processes mainly due to the initialization overhead. The fact that no duplicate elimination across partitions is performed has an impact on the amount of generated duplicates as larger numbers of partitions come with higher percentages of duplicates. However, the rate of duplicate percentage increment (initially super-linear) gradually falls as the number of partitions increases. Moreover, copying schema triples to all partitions has an impact on the amount of duplicates since, for the

same number of processes, by doubling the size of the dataset the percentage of duplicates seems to halve. It is also noted that duplicates can be reduced by excluding several rules that result in trivial inferences.

**Logic:  $\rho$ df.** Heino et al. [37] study RDFS reasoning on top of a shared memory architecture for a subset of RDFS called the  $\rho$ df vocabulary. More specifically, parallel reasoning on top of GPUs (Graphics Processing Units) is considered using the OpenCL programming model. The applicability of parallel reasoning over OpenCL using CPUs is studied as well, since the same operation can be performed both on GPUs and CPUs. Initially, the given RDFS dataset is transformed into 64-bit integers, which allows for simpler implementation, reasoning speedup coming from fast integer comparisons, and data compression which reduces required memory bandwidth. Data are stored and indexed internally providing an efficient access to the triples. A rule application sequence is defined, ensuring soundness and completeness, in order to infer all possible conclusions in several passes, where a global synchronization step is required between passes. Prior to a rule application the number of results is unknown. However, the result buffers in OpenCL must be allocated before the actual application of the rule. Thus, authors propose a solution that is based on two passes for conclusion inferences, the first pass computes the number of conclusions and the second pass materializes new triples having the required space in the result buffers.

It is pointed out that an excessive number of duplicates is generated, producing over ten times more duplicates than unique triples. Thus, two strategies are devised in order to reduce generated duplicates, namely *global* and *local strategy*. Global strategy prevents duplicates that may be generated by the application of different rules. This is achieved by indexing already derived conclusions. However, global strategy is not performed in parallel on the GPU itself but serially on the host. On the other hand, local strategy prevents duplicates that are generated during the computation of a given rule (note that each rule is computed separately). Thus, such duplicates are detected on the GPU prior to copying newly inferred conclusions to the host. In addition, in order to fully utilize the available parallelization on the GPU, authors optimize memory access such that each thread is executing the same set of instructions, namely materializes at most one triple.

Experimental evaluation showed the performance of the proposed approach over various dataset sizes and numbers of used cores, and the effect of global and local duplicate removal strategies. Authors initially compare their approach with related work on two tests showing speedups of 2.6 and 9.5 respectively. Subsequently, scalability is tested, using CPUs, over various subsets of given datasets (scaling dataset size) and various numbers of compute unit. The approach scales linearly up to 16 compute unit while for 32 compute units the performance degrades. In addition, the system is evaluated over GPUs in order to test all possible combinations of incorporating global and local duplicate removal strategies. On the tested datasets, local strategy provides over 10% speedup considering closure time and reduces the amount of generated duplicates by up to 13 times. Combining both

strategies provides less speedup compared to the local strategy, but reduces further the overall produced duplicates. Finally, on one of the two datasets, global strategy does not provide any speedup while it leads to a decrease of duplicates of less than 1%.

**Logic: RDFS.** *4sr* [38] (see also *4sr* [39] and *4s-reasoner* [40]) is a tool implemented on top of a clustered RDF triple store called *4store* [41]. Two reasoning approaches are discussed, namely forward chaining where reasoning is performed prior to any query is submitted to the system, and thus, the complete knowledge is stored, resulting in improved query response time. However, the full materialization may produce an excessive amount of data while any update to the dataset comes at the cost of recomputing the whole closure. On the other hand, *4rs* is based on backward chaining, where reasoning is performed during query time thus keeping the knowledge base relatively small on the cost of longer query response. In addition, hybrid approach is mentioned as well as the combinations of forward and backward chaining. *4rs* supports reasoning over the *Minimal* RDFS semantics by altering the architecture of *4store* and by incorporating the reasoning process into the function that returns all matching triples.

In particular, instance triples are divided into a set of non-overlapping segments that are distributed across storage nodes. Authors study the dependencies between the *Minimal* RDFS semantics rules, forming the rule chain tree. It is noted that for the application of each rule, at most one premise matches instance triples while all other premises match schema triples. This observation allows to retain the main scalability attribute of the bind (query matching) operation of *4store*, namely bind operation must run on each storage node in parallel. Thus, by replicating schema triples in each storage node, bind operation can be performed on each node independently. In order to maintain a consistent copy of schema closure, schema triples are stored and updated in a processing node called RDFS Sync, which synchronizes the schema knowledge across all storage nodes. Reasoning is incorporated into the bind operation by introducing a new binding function that returns both explicit (existing) and implicit (entailed) knowledge during query time. The new bind operation is supported by modifying several rules and maintaining two additional graphs. A detailed description of the algorithm for the new bind operation is provided.

The scalability of the system is evaluated in terms of adding more nodes and increasing the data size (from 13M to 138M triples). Preliminary work [40] reported the ability of *4rs* to handle SPARQL queries over dataset consisting of up to 500M triples. In this work, two set-ups are used in order to evaluate the performance of the system, namely a server and a cluster set-up. It is reported that, during data assertion, *4rs* comes with no additional overhead compared to *4store* since reasoning is deferred to query time. Server set-up shows good scalability when the number of used segments is less or equal to the number of available cores. In addition, it is shown that the bind operation performs worse when domain and range inference is required, while the server performs worse for the largest tested dataset. On the other hand, cluster set-up showed better results as it manages I/O operations more efficiently. For the largest used datasets, cluster reached optimal

performance while retaining linear performance, for up to 32 segments, for queries that required domain and range inference.

**Logic: RDFS.** Stream reasoning over RDF data and C-SPARQL query answering, using Yahoo S4, is studied by Hoeksema et al. [42]. Authors first introduce a naive RDFS reasoning process over S4 and point out several performance issues. Such issues include the minimization of stored triples and join operations that do not lead to inference, incompleteness of the reasoning process and duplicate elimination. Subsequently, an efficient RDFS reasoning is introduced providing the description of components for duplicate elimination, for analysis and distribution of unique triples, and components that compute RDFS rules. The process is based on eventual completeness, namely the system gradually increases its knowledge until no new knowledge can be added. In addition, reasoning operates under the assumption that schema data are inserted separately and do not constitute part of the stream.

Reasoning process feeds triples to C-SPARQL query processing. C-SPARQL queries are supported through a sequence of components that are dealing with different aspects of the given query. In particular, authors describe components that provide variable bindings for matched query patterns, perform joins on variables in the query, filter incoming bindings and emit the final results in the required format. Authors also discuss the two types of windows supported by C-SPARQL, namely a window comprises of either a fixed number of triples or a fixed period of time during which triples are entering the stream. The latter approach is adopted since it facilitates the synchronization across machines as all machines run using a synchronized timer. The former approach is more challenging as it requires precise triple counting over a distributed setting. Triples that enter the system are assigned a timestamp and an expiration time, while triples that are derived again after their expiration are reassigned their corresponding timestamps. Moreover, authors describe several components that support aggregates such as SUM, AVG, COUNT, MIN and MAX.

The system is evaluated based on two metrics, namely maximum throughput in terms of triples per second (with maximum supported throughput of 160.000 triples per second) and the number of processing nodes. When no reasoning is performed and the applied query passes through any given triple (passthrough query), high throughput is achieved even with 3 nodes showing linear performance. However, when RDFS reasoning is performed over the passthrough query, the system is showed to scale up to 8 nodes, but it is unclear why linear performance is not retained for 16 and 32 nodes. In addition, two queries are considered where no RDFS reasoning is applied. For both queries linear performance is reported for up to 8 compute nodes.

## 3.2 OWL Reasoning

**Logic: OWL Horst.** Soma et al. [43] studied how reasoning over OWL Horst knowledge

bases can be parallelized resulting in balanced workload in order to store the full materialization. Two parallelization techniques are explored, namely rule partitioning and data partitioning. Rule partitioning refers to the case where, for a given rule set, each node in the cluster is responsible for the computation of a subset of rules. Thus, the workload per rule (and node) depends both on the structure of the rule set and the number of rules. Therefore, for rule partitioning, balanced workload is difficult to be achieved. For the case of data partitioning, data are divided into subsets with each subset being assigned to a node, allowing more balanced distribution of the computation among nodes.

Authors proposed certain metrics to facilitate the evaluation of the effectiveness of each partitioning approach. *Balanced partitioning* is achieved when each processor is assigned an equal amount of work, and consequently all processors finish their work simultaneously, thus no processor remains idle (wasting computational power). Processors are coordinated by exchanging information, thus we should *minimize communication* between processors, namely each processor needs to be as independent as possible. In addition, to achieve optimal *efficiency* each conclusion must be derived by exactly one processor, thus minimizing the number of produced duplicates during the inference process. *Speed and scalability* evaluate the partitioning process itself. The chosen partitioning approach should be fast (speed) and have the potential to scale (scalability) for large datasets.

Experimental evaluation where conducted on a cluster of machines over P2P communication using Jena reasoner in order to examine both techniques. Specifically, OWL Horst closure is computed, over millions of triples. For data partitioning authors evaluated *graph* (dataset represented as a graph), *hash based* (data are assigned to nodes according to a given hash function) and *domain specific* (data distribution of the given domain is taken into consideration) *partitioning*. Results indicate that graph and domain specific partitioning have a relatively close performance as they produce balanced partitions, while a naive hash based partitioning performs badly due to imbalanced partitions. For rule based partitioning, authors point out the fact that the used rule sets were small and thus only a small number of processors could be used. Nevertheless, evaluation shows sub-linear but monotonic speedups.

**Logic: OWL Horst.** Urbani et al. [44] (see also WebPIE [45, 46]) deal with distributed RDFS reasoning using the MapReduce framework. Initially, a naive approach is presented where the application of each RDFS rule is implemented through *map* and *reduce* functions, and fixpoint iteration is required, namely all rules are applied iteratively until no new conclusion is derived. However, the naive approach suffers from shortcomings such as the excessive duplicate production and the overhead coming from repetitive rule applications. Thus, several optimizations were introduced in order to speedup the reasoning process. First, authors point out that schema triples are far less compared to instance triples, and thus, can be loaded in memory. In addition, since reasoning using MapReduce is based on term-based partitioning, several terms of a triple can be grouped together in order to reduce the amount of produced duplicates. Moreover, authors deal

with fixpoint iteration by identifying a suitable ordering for rule applications that requires a single application of each rule in order to compute the RDFS closure.

Subsequently, the complete reasoning process is described where RDFS reasoning is implemented as a sequence of MapReduce jobs. Firstly, dictionary encoding is performed where each term is replaced by a unique 8-byte identifier, while schema triples are extracted as well. Once the dataset is transformed into a more compact form, four MapReduce jobs implement the entire RDFS reasoning process. The first job deals with rules considering sub-properties, a detailed algorithm shows that within a single job, more than one rules can be applied by setting appropriately the key and the value within the *map* function. In addition, schema triples are loaded in memory for every job in order to optimize join applications. The second job describes the application of rules for domain and range. The third and the fourth job apply a duplicate elimination and rules for sub-class derivation respectively. It should be noted that rules that do not require a join operation are not included in the reasoning process as they can be applied at any point.

Experiments were performed using the Hadoop MapReduce framework in order to evaluate the efficiency of the proposed approach. Authors used several real-world datasets in order to illustrate the behavior of the system. Distributed RDFS reasoning shows good scalability properties for up to 64 nodes. However, it is pointed out that performance decreases when reasoning is performed using more than 16 nodes mainly because of the platform overhead coming from Hadoop infrastructure (each *map* and *reduce* function comes with an initialization overhead which is not compensated for small subsets of the given dataset). For the input that combined all existing datasets, containing 865M triples, 30B conclusions were derived reaching a throughput of 8.77 million triples/sec. for the output and 252.000 triples/sec. for the input (when dictionary encoding overhead is excluded). However, if the time for dictionary encoding is included as well, then the throughput falls to 4.27 million triples/sec. for the input and 123.000 triples/sec. for the output. It is also evident that the most expensive operation is the computation of rules that consider sub-classes. Finally, preliminary results for OWL reasoning indicated that further optimizations are required in order to extend the existing approach to OWL reasoning.

**Logic: OWL Horst.** WebPIE [47] (see also WebPIE [45, 46]) is an inference engine that extends a previously existing approach for distributed RDFS reasoning [44] to distributed reasoning under the OWL Horst semantics, using the MapReduce framework. Reasoning over OWL Horst introduced several challenges such as the absence of a well-defined rule application ordering, namely reasoning process is based on fixpoint iteration. Thus, duplicate derivations are increased since different rules may lead to identical conclusions during the iteration. In the work of Urbani et al. [44] each rule contained at most one instance triple, which allowed for optimization by loading schema triples in memory. However, OWL Horst rules require joins between multiple instance triples, thus introducing load balancing challenges. In addition, OWL Horst reasoning requires multiple joins

for a single rule, as opposed to RDFS reasoning where each rule required at most one join.

In the light of the aforementioned challenges, a set of newly introduced optimizations was essential in order to retain scalability. It should be noted that WebPIE [47] incorporates reasoning over RDFS as presented by Urbani et al. [44] and extends it by providing an algorithm for the fixpoint iteration. Authors apply an efficient algorithm for the calculation of transitivity. In essence, transitivity is computed in steps, with the used algorithm reducing both complexity and the number of produced duplicates. In addition, the severe load balancing problems that are caused due to *owl:sameAs* statements are dealt by finding groups of synonyms, where all resources in the group are represented by their canonical representation (group key). Subsequently, each resource in the dataset is replaced by its corresponding group key. For rules that require multiple joins using both schema and instance triples, authors devise an optimization that is based on early filtering. More specifically, instance triples are joined with all schema triples during the *map* phase, thus minimizing the number of triples that are passed to *reduce* phase. In addition, combining instance and schema triples improves load balancing since partitioning can be performed on a more general key, with each key being a combination of more than one triple terms.

Experimental evaluation showed that WebPIE has the ability to scale up to 100 billion triples, utilizing up to 64 nodes. In particular, two real-world and one benchmark datasets were used, with real-world datasets consisting of 1.51 billion and 0.91 billion triples respectively, while the benchmark dataset was used for up to 100 billion triples. In terms of scalability, WebPIE shows linear performance from 8 to 64 nodes, as well as from 1 to 100 billion triples on the benchmark dataset. It is noted that throughput is higher for larger datasets as the startup overhead coming from the platform is amortized by the amount of work. The throughput for the benchmark dataset is approximately 10 times higher compared to the real-world datasets. This result is explained by the lower complexity of the benchmark dataset, which requires less iterations in order to reach a fixpoint.

**Logic: RDFS, OWL Horst, OWL 2RL.** SAOR [48] presents various optimizations for large-scale, rule-based reasoning, materialization approaches. More specifically, the so called “partial-indexing” approach is based on the separation of TBox and ABox. Authors focus on rule sets containing only one recursive pattern. In this way, the non-recursive part is pre-computed, while the recursive part is handled through a triple-by-triple stream. Partial indexing is based on the premise that for semantic data, TBox is relatively small and non-recursive. Thus, TBox closure can be pre-computed and used during ABox reasoning.

In addition to TBox and ABox separation, authors propose optimizations over the given rule set. Using existing TBox patterns of the given rule set, rules can be rewritten with the new *templated rules* incorporating the TBox, and thus, no longer requiring TBox bindings during ABox reasoning. Once *templated rules* are computed, they can be further optimized by (a) merging equivalent templated rules, (b) constructing a rule index that matches a given input triple to the corresponding rule application, (c) recognizing rule dependencies where the application of a rule may or will lead to the application of another



rule in the new rule set, and (d) enforcing rule dependencies by always applying rules that depend on a previously applicable rule. The aforementioned optimizations have been shown to be applicable to rule sets such as RDFS, OWL Horst and OWL 2RL. Authoritative reasoning is included in the proposed approach as well. Authoritative reasoning refers to knowledge derivation where a given source is actually defining the semantics of concepts used in the body of the applied rule.

Experimental evaluation is performed over 1.12 billion quads that included provenance using a subset of OWL 2RL/RDFS axiomatic rules. The extraction of 1.1 million TBox triples took 8.2 hours on a single machine, which was reduced to 1.12 hours on 8 machines. The fastest approach proved to be the one separating TBox and ABox reasoning, while linking and merging equivalent templated rules. In general, reasoning process showed linear performance for up to 8 machines, where input of 1.12 billion triples, using 8 machines, led to the derivation of 1.58 billion triples in 3.35 hours.

### 3.3 Description Logic Reasoning

**Logic:** *ALC*. *Deslog* [49] is a parallel tableau-based description logic reasoner for the description logic *ALC*, designed for thread-level parallelism. It is utilizing the inherent non-determinism of description logic tableaux in order to implement parallel TBox classification on a shared-memory setting. Such non-determinism comes from disjunctions and qualified cardinality restrictions. The system consists of several components such as the *pre-processing layer* that translates given OWL ontology into the internal representation, the *reasoning engine layer* that performs description logic reasoning by providing services such as consistency checking, concept satisfiability, subsumption and classification, the *post-processing layer* that processes results and the *infrastructure layer* that supports auxiliary operations.

Authors point out the fact that naive tree structures are not well placed for a shared-memory setting and introduce optimized data structures. Thus, a list-based structure called *stage* and a queue-based structure called *stage pool* are used for the storage of each non-deterministic branch and all branches in the tableau, respectively. Several optimization techniques, such as lazy-unfolding, axiom absorption, semantic branching, dependency directed backtracking and model merging, were studied for their suitability for a parallel implementation and were incorporated into the system. In addition, the introduced overhead due to thread management is minimized by using efficient data structures. Such data structures allow parallel reasoning while minimizing the frequency of simultaneous access to shared data by multiple threads.

Experimental results indicate good scalability properties for TBox classification based on a multi-threading shared-memory model that is implemented in Java on a 16-core computer. Various datasets, of relatively small size, were considered with the number of axioms in each tested dataset ranging from 45 to 1140. It is shown that for small inputs

the overhead coming from parallelization due to threads manipulation and access to shared data affects significantly the performance. However, for larger inputs parallelization shows linear performance for up to 16 threads, while authors stress the fact that reasoning performance remains stable for up to 32 threads, indicating that the system could potentially utilize a larger hardware setting. Opposite to the results for the shared-memory setting, the system did not perform well on a distributed setting as the maximum speedup was below 3 even when 16 processors were assigned on a high-performance computing cluster.

**Logic:** *SHN*. *UPPR* (*Ulm University Parallel Reasoner*) [50] parallelized a sequential algorithm for *SHN* ABoxes. It is implemented in C++ and designed for shared memory settings. The reasoner parallelized the tableau procedure itself, by utilizing concurrent computation of the nondeterministic choices. *UPPR* supports nondeterministic rules such as the disjunction rule and the number restriction merge rule. Multiple alternatives can be generated by such rules and thus processed in parallel. The system utilizes a *work pool* design in which a predefined number of threads is used for processing available branches. Classification process is reduced into a unsatisfiability problem, where the process stops either when a complete tableau is found or when every possible alternative has been evaluated.

The need for a depth-first evaluation order is discussed since it favors ABoxes that may lead faster to a complete tableau. A priority queue is used, with each application of a nondeterministic rule creating ABoxes of higher priority. Opposite to the priority queue, it is mentioned that queue and stack data structures would not enforce a depth-first evaluation order. Concepts and expressions are represented internally as an array of integer arrays in order to minimize required memory. In addition, various optimizations such as naming, lazy unfolding, lexical normalization, semantic branching, simplification and caching were incorporated into the system, providing higher efficiency.

Experimental evaluation was performed on several hardware settings showing good speedups for systems consisting of up to 4 cores. On the other hand, results on a server with 24 cores did not provide significantly higher speedups possibly because the server was not used exclusively for the experiments and due to the overhead coming from frequent access to the work pool. Testing maximum cardinality restriction generated high number of alternatives, thus revealing that the overhead coming from the synchronization of the used work pool may constitute a bottleneck. Similar speedup patterns were observed for the case of a realistic ontology. The evaluation of disjunction of eight concepts showed step-wise speedups for increasing number of workers. Finally, the reasoner is reported to perform better when all optimizations are enabled.

**Logic:** *ALCNH<sub>R+</sub>*. Liebig et al. [51] propose an approach, based on their previous work of Liebig et al. [50], for parallelizing a sequential algorithm for *ALCNH<sub>R+</sub>* ABoxes, while extending it for *SHIQ*. It has been implemented in C++, initially designed for shared memory settings but with the ability to run on a distributed setting as well. The reasoning task is reduced to a corresponding ABox unsatisfiability problem for

which authors have implemented a tableau prover. Two main approaches for reasoning parallelization are discussed, namely parallelization on either *reasoner level* or *proof level*. Reasoner level refers to parallelization by running simultaneously several instances of a given reasoner. However, it is inefficient as such instances do not exchange information effectively, leading to repetitive computations. On the other hand, parallelizing on proof level allows to evaluate independent proof steps concurrently, thus allowing to utilize all available processing cores while increasing the possibility of a good guess that will lead to a solution. The system follows the proof level parallelization approach.

Three rules are reported to have inherent nondeterminism, namely disjunction rule, number restriction merge rule and choose rule. Such rules generate alternatives that may be evaluated in parallel. As in work of Liebig et al. [50], reasoner relies on a *work pool* design, where each *work unit* defines a fixed number of *work executors*. Each work unit contains a *work controller* that controls the *work queue* and communicates with the work executors within the work unit. Each work executor performs reasoning over an alternative in the (work) queue, generating new alternatives, if necessary, and reporting whether a solution was found or all available alternatives have been exhausted (where no satisfiable alternative was found). The work queue is organized using a custom priority queue, namely alternatives of priority  $n$  generate alternatives of priority  $n + 1$ , thus promoting a depth-first search in order to speed up the reasoning process. The system allows reasoning over a distributed setting as well, with *work distributor* being a coordinator of several work units that balances the work load in the cluster. Experimental evaluation was deferred to future work.

**Logic:** *ALCHL*. MapResolve [52] studies the challenges of scalable reasoning that is based on the MapReduce framework. The main challenge lies on the fact that in order to compute the closure, a sequence of MapReduce jobs is required as newly derived knowledge must be fed back to the system for further derivations, thus leading to repeated inferences. This issue is identified in existing approaches, for RDFS and OWL Horst materialization, and  $\mathcal{EL}^+$  classification, which are based on MapReduce. Authors present a distributed resolution method for checking the satisfiability of *ALCHL*, while the same method can be applied to first order theories as well. The provided naive approach for distributed description logic resolution points out the problem of repeated inferences.

To overcome this challenge, authors follow a method where separate sets are maintained for clauses that have already been evaluated and clauses that still require rule application. At each MapReduce job, usable clauses are allocated during the map phase, while already evaluated clauses are loaded to each reducer for resolution. The reducer also deletes duplicate clauses and clauses that are subsumed by other clauses so as to further improve efficiency. At the end of the resolution process, both sets are stored for the next job. This process is repeated until the whole closure is computed. In addition, authors deal with load balancing by distributing the set of usable clauses evenly among reducers. The number of usable clauses that are assigned to each reducer is adjusted after each job by

taking into account the predicted and the actual runtime. No experimental evaluation was provided, thus it is unclear whether the minimization of repeated inferences is able to amortize the cost of storing and parsing the sets of clauses in every job.

**Logic:**  $\mathcal{ELH}_{\perp, \mathcal{R}+}$ . Ren et al. [53] extend an existing parallel TBox reasoning for  $\mathcal{ELH}_{\mathcal{R}+}$ , to parallel ABox reasoning algorithm for  $\mathcal{ELH}_{\perp, \mathcal{R}+}$ . The proposed algorithm supports the bottom concept ( $\perp$ ) in order to model disjointness and inconsistency. Several optimizations are introduced in order to increase the efficiency of parallel ABox reasoning. In particular, authors study the set of completion rules and show that reasoning process can be separated into three steps, namely reasoning over: (a) TBox completion rules, (b) relation completion rules, and (c) type completion rules. In this way, each step provides a pre-computed subset of conclusions to the next step, thus minimizing overheads and allowing for highly efficient reasoning at each step.

A parallel algorithm for saturation of axioms under the given inference rules is presented. Initially, a set of axioms is given as input, with each axiom being processed, in parallel, thus initializing and activating the corresponding context queue, where a context is a common concept for a given rule. Each active context is processed, in parallel, applying rules by using its corresponding axioms. The reasoning process is completed when all active contexts have been processed. Experimental evaluation shows that the proposed algorithm has the capacity to compute all ABox entailments for an ontology containing 1 million individuals and 9 million axioms in approximately 3 minutes.

**Logic:**  $\text{DL-Lite}_{\mathcal{R}}$ . Fokoue et al. [54] present an approach for querying over  $\text{DL-Lite}_{\mathcal{R}}$  ontologies that are stored over multiple data sources, by applying distributed summarization. More specifically, authors point out that existing work on ABox summarization could not be applied in a decentralized setting as it requires complete explicit information for each individual. Thus, a distributed summarization approach is proposed, which is based on two steps. First, the construction of each summary is performed locally, and then, the generated local summaries are merged in a central location.

Distributed summarization is in line with how the Linked Open Data is organized. Local sources can decide on the type of the hash function that will be applied to individuals in the knowledge base, and thus, define the number of generated buckets, which allows them to strike a balance between the precision and the size of the summary. In addition, a normalization mechanism is required in order to allow local summaries to be merged in a balanced fashion. Finally, the hash value of a given individual should be assigned to the most authoritative source. The aforementioned requirements can be modeled for Linked Open Data by handling carefully the URI of each individual.

Local summaries enabled optimizations such as query pruning, where an empty local summary can be used in order to ensure an empty result set for the given query, and efficient source selection where only relevant sources are evaluated over the given query. Distributed summarization can eliminate distributed joins if all required terms for the join operation belong to the same source or matching terms, for a given join, are collocated on

the same source. Experimental evaluation shows that the summary of a given data source can range from 0.037% to 9.2%, depending on the number of similar concepts in each source, while the cost of generating the distributed summary is amortized by the efficient query execution for the majority of the tested datasets and engines.

**Logic: *SHOIN* (OWL DL).** Du et al. [2] proposed a decomposition-based approach that allows the optimization of conjunctive query answering in Description Logic *SHOIN* (also known as OWL DL). The underlying idea is based on an initial computation of explicit answers, namely answers/facts that satisfy the given query. Moreover, candidate answers and target ontologies are identified so as to check, using existing reasoners, only ontologies that could lead to answers to the given query (by checking whether candidate answers are indeed answers to the given query). Authors point out that while target ontologies do not have common ABox axioms, they may have common TBox axioms.

DecomBaR (**D**ecomposition-**B**ased **R**easoner) implements the proposed method showing advantages over existing reasoners. Specifically, experimental evaluation showed that DecomBaR performs well when it does not generate any candidate answers since it can compute answers by accessing its database through SQL queries. In addition, authors mention the scalability advantages of DecomBaR over existing reasoners due to its design, which is based on SQL queries when no candidate answers are generated, and on evenly distributed ABox axioms across extracted ontologies, for queries that generate candidate answers. Finally, DecomBaR is shown to be more scalable for increasing TBox complexity over existing reasoners, while having been evaluated over ontologies containing millions of ABox axioms.

### 3.4 Conclusion

Traditionally, the area of knowledge representation has focused on complex knowledge structures and reasoning methods for processing these structures. The new arising challenge is to study how reasoning can process such interesting knowledge structures in conjunction with huge data sets. To fully exploit the immense value of such datasets and their interconnections, one should be able to reason over them using rule sets that would allow the aggregation, visualization, understanding and exploitation of the raw data that comprise the databases. Such reasoning is based on rules which capture the inference semantics of the underlying knowledge representation formalism, but also rules which encode commonsense, practical knowledge that humans possess and would allow the system to automatically reach useful conclusions based on the provided data and infer new and useful knowledge based on the data.

In this work, we consider nonmonotonic rule sets [55, 56]. Such rule sets provide additional benefits because they are more suitable for encoding commonsense knowledge and reasoning. In addition, nonmonotonic rules avoid triviality of inference, which could easily occur when low-quality raw data is fed to the system; the latter is common in

this setting, given the interconnection of data from different sources, over which the data engineer has no control.

The main challenge rising in such a setting is the feasibility of reasoning over such large volumes of data. One of the most promising methods to address this problem is by using massively parallel reasoning processes that would handle reasoning by using several computers in the cloud, assigning each of them a part of the parallel computation.

As discussed above, there has been significant progress in parallel reasoning, scaling reasoning up to 100 billion triples [47]. Nevertheless, current approaches have been restricted to monotonic reasoning, namely RDFS and OWL horst, or have not been evaluated for scalability [57]. However, in many application scenarios, one needs to deal with poor quality data (e.g., involving inconsistency or incompleteness), which could easily lead to reasoning triviality when considering rules based on monotonic formalisms; this problem can be managed with nonmonotonic rules and nonmonotonic reasoning.

## Chapter 4

# Defeasible Logic

In this chapter, an approach for single variable rule sets is proposed, followed by an extension where stratified rule sets of arbitrary arity are considered. Finally, the challenges of the general approach for non-stratified rule sets are discussed, while experimental evaluation shows the feasibility of the proposed approach.

### 4.1 Single Variable Rule Sets

*Single variable* rule sets refer to rule sets where each rule contains only one variable. As a running example, let us consider the following rule set:

$$\begin{aligned} r1 &: \text{bird}(X) \rightarrow \text{animal}(X) \\ r2 &: \text{bird}(X) \Rightarrow \text{flies}(X) \\ r3 &: \text{brokenWing}(X) \Rightarrow \neg\text{flies}(X) \\ r3 &> r2 \end{aligned}$$

In this simple example we try to decide whether something ( $X$ ) is an animal and whether it is flying or not. Given the facts  $\text{bird}(\text{eagle})$  and  $\text{brokenWing}(\text{eagle})$ , as well as the superiority relation ( $r3 > r2$ ), we conclude that  $\text{animal}(\text{eagle})$  and  $\neg\text{flies}(\text{eagle})$ . Note that each rule has only one variable ( $X$ ), while an additional variable (say  $Y$ ) in any rule would change the classification of the rule set from single to multiple variable (see Section 4.2).

Taking into account the fact that every rule has only one variable, 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 Figure 4.1. Equivalently, we can view this process as performing reasoning on the rule set:

```

map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (a fact)
  String argument = getArguments(value);
  String predicate = getPredicate(value);
  emit(argument, predicate);

reduce(String key, Iterator values):
  // key: argument
  // values: list of predicates (facts)
  Set facts =  $\emptyset$ ;
  for all value  $\in$  values do
    facts.add(value);
  end for
  Reasoner reasoner = Reasoner.getCopy();
  reasoner.applyReasoning(facts);
  emit(key, reasoner.getResults());

```

Figure 4.1: Single variable inference

$$\begin{aligned}
r1 &: \text{bird} \rightarrow \text{animal} \\
r2 &: \text{bird} \Rightarrow \text{flies} \\
r3 &: \text{brokenWing} \Rightarrow \neg\text{flies} \\
r3 &> r2
\end{aligned}$$

for each unique argument value.

As far as MapReduce is concerned, *Map* function reads facts of the form  $\text{predicate}(\text{argument})$ , extracts the argument ( $\text{argument} = \text{getArgument}(\text{value})$ ) and predicate ( $\text{predicate} = \text{getPredicate}(\text{value})$ ) from each fact ( $\text{value}$ ), and emits pairs of the form  $\langle \text{argument}, \text{predicate} \rangle$ .

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

$$\begin{aligned}
&\langle \text{eagle}, \text{bird} \rangle \\
&\langle \text{owl}, \text{bird} \rangle \\
&\langle \text{pigeon}, \text{bird} \rangle \\
&\langle \text{eagle}, \text{brokenWing} \rangle \\
&\langle \text{owl}, \text{brokenWing} \rangle
\end{aligned}$$

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:

$$\begin{aligned}
&\langle \text{eagle}, \langle \text{bird}, \text{brokenWing} \rangle \rangle \\
&\langle \text{owl}, \langle \text{bird}, \text{brokenWing} \rangle \rangle \\
&\langle \text{pigeon}, \langle \text{bird} \rangle \rangle
\end{aligned}$$



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 the reasoner (described later on) gets as input a list of predicates (extracted from *values*) 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.

- *eagle* having *bird* and *brokenWing* as facts, deriving *animal(eagle)* and  $\neg$ *flies(eagle)*
- *owl* having *bird* and *brokenWing* as facts, deriving *animal(owl)* and  $\neg$ *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 by Maher [58]. Prior to any *Reduce* function is applied, given rule set must be parsed initializing indexes and data structures required for reasoning. Although implementation details of the reasoner are out of the scope of this work, we will explain all the functions used in Figure 4.1.

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.applyReasoning(facts)* performs reasoning on each copy, receiving as input the corresponding list of predicates (*facts*). Derived data are stored internally by each copy of the reasoner. The extraction of the derived data is performed by the *reasoner.getResults()*.

**Soundness and Completeness.** The algorithm for single variable rule sets is sound and complete since it performs reasoning using every given fact (*Map*). This data partitioning (group/sort) 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 (further details can be found in the work of Maher [58]) for each unique value (*Reduce*). Thus, maximal and valid data derivation is assured.

**Computational complexity.** Complexity analysis is based on the number of facts given as input (say  $n$ ), with the worst case being where each fact belongs to a different group, namely has a unique value for its argument. During *Map* each fact is transformed into a new key/value pair, thus *Map* has complexity  $O(n)$ . Group/sort phase is based on quicksort algorithm, thus having a complexity of  $O(n \log n)$ . *Reduce* is deriving conclusions for all predicates in the given rule set (say  $k$ ), with defeasible logic algorithm having linear complexity [58], thus *Reduce* has complexity  $O(k*n)$ , where  $k$  is significantly smaller than  $n$  and could be considered as a constant. Overall the complexity for single variable rule sets is  $O(n \log n)$ .

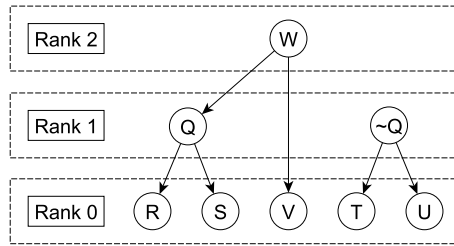


Figure 4.2: Stratified rule set. Predicates are assigned to ranks.

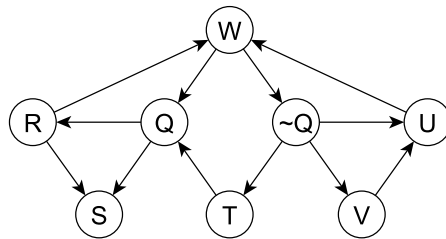


Figure 4.3: Non-stratified rule set. Predicates cannot be assigned to ranks.

Note that single variable rule sets are a special case of multi variable rule sets, coming with a certain advantage. Since each rule has only one variable, both premises and conclusions for a given (unique) argument value will be collocated at the same processing node (reducer). Thus, for single variable rule sets “global” fixpoint is reached within one MapReduce job (where “local” fixpoint refers to the derivation of every possible conclusion, for each unique value, by a reducer).

## 4.2 Multi Variable Rule Sets

Rule sets containing rules with an arbitrary number of variables are called *multi variable* rule sets. In this section, we first consider a restricted form of multi variable rule sets namely *stratified* rule sets, for which we provide a parallel solution based on the MapReduce framework, and then outline arising challenges for *non-stratified* rule sets.

According to the Definition 2.1.1, for a stratified rule set, predicates that are not supported by any rule are assigned rank 0. Subsequently, predicates depending on predicates of rank 0, are assigned rank 1. In general, predicates depending on predicates of rank up to  $k - 1$ , are assigned rank  $k$ . Thus, stratification is achieved when all predicates are assigned a rank, provided that complementary predicates are assigned to equal ranks. For non-stratified rule sets, such stratification process is inapplicable.

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:  $Q(X,Z), V(Z,Y) \Rightarrow W(X,Y)$ .  
 r1 > r2.

*Stratified theories* are theories based on stratified rule sets. Stratified theories are often called decisive in the literature [59]. Figure 4.2 depicts how predicates are assigned to ranks for the above-mentioned stratified rule set, while Figure 4.3 depicts predicate dependencies of a non-stratified rule set.

**Proposition 1** [59] *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  $+\partial$  as well. 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 rank  $k - 1$ , and a rule whose head is assigned rank  $k$  and body containing literals of rank up to  $k - 1$  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 ranks.

### 4.2.1 Reasoning Overview

In this subsection we provide an overview of the defeasible reasoning process for the case of stratified multi variable rule sets. Pseudo-code depicting the overall reasoning process is provided in Figure 4.4.

Defeasible reasoning process requires as input a stratified rule set and a set of facts (reasoningProcess(*rules*, *facts*)). Our first step is to process the given rule set (applyStratification(*rules*)) in order to retract the corresponding *ranks*. As *ranks*, we refer to a structure that contains the corresponding predicates for each rank as well as the given rules for each predicate.

During reasoning we will use the representation ( $\langle \text{fact}, (+\Delta, +\partial) \rangle$ ) to store our inferred facts. Thus, we begin by transforming the given facts, in a single MapReduce pass, into ( $\langle \text{fact}, (+\Delta, +\partial) \rangle$ ).

```

reasoningProcess(rules, facts):
  // rules: stratified rule set
  // facts: set of facts
  ranks = applyStratification(rules);
  KB = initialPass(facts);
  for i=1; i <= ranks.getMax(); i++ do
    tmp = pass_#1(KB, ranks.get(i));
    new = pass_#2(KB, tmp, ranks.get(i));
    KB = KB ∪ new;
  end for

```

Figure 4.4: Overall reasoning process

Now let us consider for example the facts  $R(a,b)$ ,  $S(b,b)$ ,  $T(a,e)$ ,  $U(e,b)$  and  $V(b,c)$ . The *initial pass* ( $\text{initialPass}(\text{facts})$ ) on these facts will create the following output:

$$\begin{aligned}
&\langle R(a,b), (+\Delta, +\partial) \rangle \\
&\langle S(b,b), (+\Delta, +\partial) \rangle \\
&\langle T(a,e), (+\Delta, +\partial) \rangle \\
&\langle U(e,b), (+\Delta, +\partial) \rangle \\
&\langle V(b,c), (+\Delta, +\partial) \rangle
\end{aligned}$$

resulting in our initial knowledge base ( $KB$ ).

No reasoning needs to be performed for the lowest rank (rank 0) since these predicates ( $R, S, T, U, V$ ) do not belong to the head of any rule. Considering the definition of  $+\partial$ ,  $-\partial$ , defeasible logic introduces uncertainty regarding inference, since certain facts/rules may “block” the firing of other rules. This can be prevented if we reason for each rank separately, starting from rank 1 and continuing to higher ranks. Thus, for a hierarchy of  $N$  ranks ( $\text{ranks.getMax}()$ ) we have to perform  $N - 1$  times the procedure described below.

A key difference, compared to single variable rule sets, is that applicable rules for a given literal (or its complement) may be computed, concurrently, by different nodes in the cluster, while derivation requires all the available information, for a given literal (and its complement), to be collocated on the same node. Thus, in order to perform defeasible reasoning we have to break up the process and run two passes for each rank.

The first pass ( $\text{pass\_}\#1(KB, \text{ranks.get}(i))$ ) computes all applicable rules for rank  $i$ , based on our current knowledge base ( $KB$ ), and the corresponding predicates and rules for rank  $i$  ( $\text{ranks.get}(i)$ , for details see Section 4.2.2). The second pass ( $\text{pass\_}\#2(KB, \text{tmp}, \text{ranks.get}(i))$ ) performs the actual reasoning, based on our current knowledge base ( $KB$ ), the applicable rules for rank  $i$  ( $\text{tmp}$ ), and the corresponding predicates and rules for rank  $i$  ( $\text{ranks.get}(i)$ ), computing for each literal whether it is definitely or defeasibly provable (for details see Sections 4.2.3).

Let us illustrate why reasoning has to be performed for each rank separately. Consider

a scenario where we perform defeasible reasoning, on the aforementioned rule set, for all ranks simultaneously. Now, we may no longer claim that if a rule supporting  $W(X, Y)$  is not applicable then it will not be applicable in subsequent passes. Indeed, reasoning about  $W(X, Y)$  will “block” until a final conclusion on  $Q(X, Y)$  and  $\neg Q(X, Y)$  is derived. Thus, defeasible reasoning process will be applied following an implicit stratification. By applying stratification explicitly prior to reasoning, we minimize overheads as every possible fired rule or derived conclusion is calculated once.

### 4.2.2 Pass #1: Fired Rules Calculation

During the first pass, we calculate all applicable rules for a given rank, based on techniques used for basic and multi-way joins as described in the work of Afrati et al. [60]. 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 joins are performed on common argument values. Consider the following generic rule (whose calculation follows the pseudo-code in Figure 4.5):

$$r : A(X, Z), B(Z, Y) \{ \rightarrow, \Rightarrow, \rightsquigarrow \} [-] P(X, Y).$$

Let us elaborate on the used notation in Figure 4.5. In case the variable *value* contains a literal of the relation  $A(X, Z)$ , then *value.pred* refers to the predicate  $A$ , *value.X* refers to the argument  $X$ , *value.Z* refers to the argument  $Z$ , and *value.tag* refers to a subset of  $\{+\Delta, -\Delta, +\partial, -\partial\}$  that defines our knowledge about the provability of the literal. For a given rule  $r$ , *r.arrow* refers to the arrow of the rule ( $\rightarrow, \Rightarrow, \rightsquigarrow$ ), which defines the type of the rule (strict, defeasible, defeater). Considering the conclusion (see *Reduce* function),  $[-]$  denotes that “ $\neg$ ” is optional, allowing to represent and derive both positive and negative literals.

The key observation for a basic join is that relations  $A$  and  $B$  can be joined on their common argument  $Z$ . Based on this observation, during *Map* operation we emit pairs of the form  $\langle Z, (A, X) \rangle$  for predicate  $A$  and  $\langle Z, (B, Y) \rangle$  for predicate  $B$ . The idea is to join  $A(X, Z)$  and  $B(Z, Y)$  only for literals that have the same value on argument  $Z$ . During *Reduce* operation we combine  $A(X, Z)$  and  $B(Z, Y)$  producing  $P(X, Y)$ .

Such joining process is tailored for monotonic logics where conclusion derivation is performed immediately. However, in order to support reasoning under defeasible logic this approach must be extended. We must record all fired rules prior to any conclusion inference since this information is required by the defeasible logic derivation.

During *Map* operation we must additionally pass the *value.tag*, while during *Reduce* operation after going through all *values* (first *for* loop), we match premises (second *for* loop) and emit both the consequent ( $P(a.X, b.Y)$ ) and the *knowledge* about the fired rule. For a strict rule ( $r\_isStrict = true$ ) with all premises being definitely provable ( $+\Delta a$  and  $+\Delta b$ ) our derived *knowledge* is  $[-], +\Delta, +\partial, r$ , while otherwise our derived *knowledge* is

```

map(Long key, String value):
// key: position in document (irrelevant)
// value: document line (derived conclusion)
if value.pred == "A" then
    emit(value.Z, {value.pred, value.X, value.tag});
else if value.pred == "B" then
    emit(value.Z, {value.pred, value.Y, value.tag});
end if

reduce(String key, Iterator values):
// key: matching argument
// values: literals for matching
List a_List =  $\emptyset$ , b_List =  $\emptyset$ ;
bool r_isStrict = (r.arrow == "→")? true : false;
for all value  $\in$  values do
    if value.pred == "A" then
        a_List.add({value.X, value.tag});
    else if value.pred == "B" then
        b_List.add({value.Y, value.tag});
    end if
end for
for all a  $\in$  a_List do
    for all b  $\in$  b_List do
        if r_isStrict and  $+\Delta a$  and  $+\Delta b$  then
            emit("P(a.X, b.Y)", "[ $\neg$ ],  $+\Delta$ ,  $+\partial$ , r");
        else
            emit("P(a.X, b.Y)", "[ $\neg$ ],  $+\partial$ , r");
        end if
    end for
end for

```

Figure 4.5: Fired rules calculation given the generic rule  $r : A(X,Z), B(Z,Y) \{\rightarrow, \Rightarrow, \rightsquigarrow\} [\neg]P(X,Y)$ .

( $[\neg], +\partial, r$ ). Setting  $[\neg]$  in the *knowledge*, instead of the consequent, is a minor optimization facilitating facts processing during *step #2* (see Section 4.2.3), as derivation process requires both  $p$  and  $\neg p$  to be sent to the same node.

Now consider again the stratified rule set described in the beginning of the section along with the produced output from the *initial pass*. As a running example for basic join, we will perform reasoning for rank 1, using as premises all the available information for predicates of rank 0, and computing rules  $r1$  and  $r2$ . The *Map* function will emit the following pairs (literals with predicate  $V$  are ignored as they do not belong to a rule with consequent of rank 1):

$$\begin{aligned} &\langle b, (R, a, +\Delta, +\partial) \rangle \\ &\langle b, (S, b, +\Delta, +\partial) \rangle \\ &\langle e, (T, a, +\Delta, +\partial) \rangle \\ &\langle e, (U, b, +\Delta, +\partial) \rangle \end{aligned}$$

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

$$\begin{aligned} &\langle b, \langle (R, a, +\Delta, +\partial), (S, b, +\Delta, +\partial) \rangle \rangle \\ &\langle e, \langle (T, a, +\Delta, +\partial), (U, b, +\Delta, +\partial) \rangle \rangle \end{aligned}$$

During the reduce phase we combine premises in order to emit the fired rule consequent and the corresponding knowledge. Thus, the reducer with key:

$$\begin{aligned} &b \text{ will emit } \langle Q(a, b), (+\partial, r1) \rangle \\ &e \text{ will emit } \langle Q(a, b), (\neg, +\partial, r2) \rangle \end{aligned}$$

As we see here,  $Q(a, b)$  and  $\neg Q(a, b)$  are computed by different reducers which do not communicate with each other. According to the MapReduce framework, communication between nodes during the map and reduce phase is prohibited. 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.

In case of multi-way joins, we gradually compute the fired rule by applying our approach for basic joins multiple times. Consider the following rule:

$$r' : A(X, Z), B(Z, W), C(W, Y) \rightarrow \neg P(X, Y).$$

and conclusions for predicates  $A$ ,  $B$  and  $C$

$$\begin{aligned} &\langle A(1, 2), (+\Delta, +\partial) \rangle \\ &\langle B(2, 3), (+\Delta, +\partial) \rangle \\ &\langle C(3, 4), (+\partial) \rangle \end{aligned}$$

As  $r'$  is a strict rule, we must keep track of whether all premises are  $+\Delta$  or not. In order to compute  $\neg P(X, Y)$ , we first join  $A(X, Z)$  and  $B(Z, W)$  on  $Z$ , producing a temporary literal (say  $AB(X, W)$ ) deriving  $\langle AB(1, 3), (+\Delta, +\partial, r') \rangle$ . Note that, thus far, all premises

```

map(Long key, String value):
  // key: position in document (irrelevant)
  // value: inferred conclusion/fired rule
  String literal = getLiteral(value);
  String knowledge = getKnowledge(value);
  if literal.pred ∈ currentRank then
    emit(literal, knowledge);
  end if

reduce(String key, Iterator values):
  // key: literal
  // values: inferred knowledge
  List knowledge = ∅;
  for all value ∈ values do
    knowledge.add(value);
  end for
  for all p in {key, ¬key} do
    if +Δp ∈ knowledge then
      continue;
    else if ∃k ∈ knowledge such that
      k.r.arrow == “→” and
      k.tag == +Δ then
      emit(p, “+Δ,+∂”);
    else if reasoning(p, knowledge) == +∂ then
      emit(p, “+∂”);
    end if
  end for

```

Figure 4.6: Defeasible reasoning

( $A(1,2)$  and  $B(2,3)$ ) are  $+\Delta$  and  $AB(X,W)$  is a positive literal. Subsequently, we join  $AB(X,W)$  and  $C(W,Y)$  on  $W$ , computing the fired rule ( $\langle P(1,4), (\neg, +\partial, r') \rangle$ ), adding “ $\neg$ ” to the *knowledge* as  $\neg P(X,Y)$  is a negative literal, and adding only  $+\partial$  as  $C(3,4)$  is a  $+\partial$  premise (not all premises are definitely provable). More optimized applications of multi-way joins, from a computational point of view, can be found in the work of Afrati et al. [60], although requiring the extension from monotonic to defeasible logic as discussed above.

### 4.2.3 Pass #2: Defeasible Reasoning

We proceed with the second pass. Once fired rules are calculated, a second pass performs reasoning for each literal separately. The pseudo-code for *Map* and *Reduce* functions, for stratified rule sets, is depicted in Figure 4.6.

After both *initial pass* and fired rules calculation (pass #1), our knowledge base will



consist of:

$$\begin{aligned} &\langle R(a,b), (+\Delta, +\partial) \rangle \\ &\langle S(b,b), (+\Delta, +\partial) \rangle \\ &\langle T(a,e), (+\Delta, +\partial) \rangle \\ &\langle U(e,b), (+\Delta, +\partial) \rangle \\ &\langle V(b,c), (+\Delta, +\partial) \rangle \\ &\langle Q(a,b), (+\partial, r1) \rangle \\ &\langle Q(a,b), (\neg, +\partial, r2) \rangle \end{aligned}$$

During the *Map* operation we first extract from *value* the *literal* (`getLiteral(value)`) and the *knowledge* about the literal (`getKnowledge(value)`). Subsequently, we check whether the predicate of the extracted literal (*literal.pred*) belongs to the rank (*currentRank*) that we currently perform reasoning for (*literal.pred*  $\in$  *currentRank*). Note that for each literal *p*, both *p* and  $\neg p$  are sent to the same reducer, with “ $\neg$ ” in *knowledge* distinguishing *p* from  $\neg p$ . The *Map* function will emit the following pairs (recall that we perform reasoning for rank 1):

$$\begin{aligned} &\langle Q(a,b), (+\partial, r1) \rangle \\ &\langle Q(a,b), (\neg, +\partial, r2) \rangle \end{aligned}$$

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

$$\langle Q(a,b), \langle (+\partial, r1), (\neg, +\partial, r2) \rangle \rangle$$

During the reduce phase we traverse over *values* in order to update our *knowledge* (`knowledge.add(value)`). Once *knowledge* contains all the available information, for the given literal (*key*), we proceed with defeasible reasoning. Note that since *key* contains the positive literal, we need to perform reasoning for both *key* and  $\neg key$ .

In case literal *p* is already contained in our knowledge base ( $+\Delta p \in knowledge$ ), due to the *initial pass*, we do not emit any output (continue) as doing so would produce duplicates. Alternatively, we check whether there is an element *k* in *knowledge* ( $\exists k \in knowledge$ ) containing the information that a strict rule ( $k.r.arrow == \text{“}\rightarrow\text{”}$ ) fired with all premises being definitely provable ( $k.tag == +\Delta$ ). In this case, we emit that literal *p* is definitely provable (`emit(p, “+Δ,+∂”)`). Our final option is to establish whether literal *p* is defeasibly provable, by performing defeasible reasoning (`reasoning(p, knowledge) == +∂`), and emit the corresponding conclusion (`emit(p, “+∂”)`). Thus, the reducer with key:

$$Q(a,b) \text{ will emit } \langle Q(a,b), (+\partial) \rangle$$

Note that since  $\neg Q(a,b)$  is not defeasibly provable ( $-\partial$ ), it is not added to our knowledge base.

#### 4.2.4 Final Remarks

As we see, the approach for stratified multi variable rule sets turns out to be far more difficult, requiring multiple passes compared to the single-pass approach for single variable rule sets. Moreover, the total number of MapReduce passes is independent of the size of the given input, but, depends on the number of ranks, namely the structure of the given stratified rule set. As mentioned in Section 4.2.1, performing reasoning for each rank separately enforces explicitly the implicit stratification of the reasoning process, while eventually, our knowledge base consists of  $+\Delta$  and  $+\partial$  literals.

**Soundness and Completeness.** The proposed method for stratified multi variable rule sets is sound and complete given that it fully complies with the defeasible logic provability. Indeed, stratification provides a well-defined reasoning sequence ensuring that performing reasoning from lower to higher ranks, provided that the reasoning process at each rank is sound and complete, leads to all derived conclusions being sound since reasoning for rank  $k$  is based on a sound and complete knowledge base for ranks up to  $k - 1$ , and a complete derivation of all possible conclusions since we enforce explicitly the implicit stratification of the reasoning process. At each rank, the first pass computes all applicable rules as *Map* processes all required knowledge for rule application, group/sort phase redirects knowledge (with no knowledge being lost), and *Reduce* computes all applicable rules for each group of matching arguments. Note that multi-way joins follow the same principle. Subsequently, the second pass processes the available knowledge for all predicates of rank  $k$  during *Map*, group/sort phase forms a group for each literal, while *Reduce* derives any possible conclusion following the defeasible logic algorithm.

**Computational complexity.** Complexity analysis is based on the number of facts given as input (say  $n$ ), based on two rules (having conclusions  $p$  and  $\neg p$ ) that are calculated by applying a basic join, for a single rank, with the worst case being where all facts belong to the same group, namely have the same value for the matching argument, but generate different conclusions, namely support different literals. Note that here we provide the intuition behind the calculation of computational analysis, while a generic formula that could model any given rule set is out of the scope of this work. The first pass computes all applicable rules as *Map* processes all required knowledge for rule application, thus *Map* has complexity  $O(n)$ . Group/sort phase is based on quicksort algorithm, thus having a complexity of  $O(n \log n)$ . *Reduce* computes all applicable rules for each group of matching arguments, while in the worst case applicable rules are computed over two lists of length  $n/2$  each, thus *Reduce* has complexity  $O(n^2)$ . Subsequently, the second pass processes the available knowledge for each  $p$  and  $\neg p$  during *Map*, namely *Map* has complexity  $O(n^2)$ . Group/sort phase is based on quicksort algorithm, thus having a complexity of  $O(n^2 \log n^2)$ . Finally, *Reduce* derives any possible conclusion for any given pair  $p$  and  $\neg p$ , where the application of the defeasible logic algorithm has linear complexity (say  $O(k)$ , with  $k$  being the number of rules for  $p$  and  $\neg p$ ), thus *Reduce* has complexity of  $O(k * n^2)$ ,

where  $k$  is significantly smaller than  $n$ , and could be considered as constant. Overall complexity for the considered rule set is  $O(n^2 \log n^2)$ .

The situation for non-stratified multi variable rule sets is more complex. Reasoning can be based on the algorithm described in the work of Maher et al. [11], performing reasoning until no new conclusion is derived. The total number of required passes is generally unpredictable, depending both on the given rule set and the data distribution. However, since propositional defeasible logic has linear complexity [58], the worst case analysis indicates that in order to derive  $N$  conclusions,  $O(N)$  MapReduce passes are required, with each pass deriving a conclusion.

Maher et al. [11] proposed a sequential approach for in-memory defeasible reasoning. Parallelizing this approach and handling large amounts of facts comes with certain challenges. There is a severe scalability challenge posed by  $-\Delta$  and  $-\partial$  (specifically part 2.1) provability. Consider the following rule:

$$r^* : P(X,Z), Q(Z,Y) \rightarrow P(X,Y).$$

In order to establish that  $P(a,b)$  is not definitely provable ( $-\Delta$ ) we need to check every possible instantiated rule, namely for every value of  $Z$  we need to check whether either  $P(a,Z)$  or  $Q(Z,b)$  is  $-\Delta$ . For the aforementioned rule ( $r^*$ ), having  $N$  constants in the given dataset we need to check  $N$  instantiated rules for every  $-\Delta$  conclusion.

In general, for  $N$  constants in the given dataset and  $k$  variables, in a given rule, that do not belong to the head of the rule (e.g., the variable  $Z$  in the aforementioned rule  $r^*$ ), every  $-\Delta$  conclusion will require  $N^k$  instantiated rules to be checked. By checking every possible instantiated rule, we introduce a significant overhead that can become prohibitive even for relatively small datasets.

In addition to this scalability challenge, we need to consider how this reasoning approach affects parallelization. More specifically, an efficient mechanism for such computation is yet to be defined since all the available information for each literal must be processed by a single node (recall that nodes do not communicate with each other), causing either main memory insufficiency or skewed load balancing decreasing the parallelization. Parallelization techniques such as OpenMP<sup>1</sup> and Message Passing Interface (MPI) may provide higher degree of flexibility than the MapReduce framework, thus allowing a more balanced distribution of the workload.

### 4.3 Evaluation

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

**Methodology.** Our evaluation is centered around scalability and the capacity of our system to handle large datasets. In line with standard practice in the field of high-

---

<sup>1</sup><http://openmp.org/wp/>

Table 4.1: Rule set

Rule ID	Rule or Superiority relation
r1	FullProfessor(X) $\rightarrow$ Professor(X).
r2	AssociateProfessor(X) $\rightarrow$ Professor(X).
r3	AssistantProfessor(X) $\rightarrow$ Professor(X).
r4	publicationAuthor(P,X), publicationAuthor(P,Y) $\rightarrow$ commonPublication(X,Y).
r5	teacherOf(X,C), takesCourse(Y,C) $\rightarrow$ teaches(X,Y).
r6	teachingAssistantOf(X,C), takesCourse(Y,C) $\rightarrow$ teaches(X,Y).
r7	commonPublication(X,Y) $\rightarrow$ commonResearchInterests(X,Y).
r8	hasAdvisor(X,Z), hasAdvisor(Y,Z) $\rightarrow$ commonResearchInterests(X,Y).
r9	hasResearchInterest(X,Z), hasResearchInterest(Y,Z) $\rightarrow$ commonResearchInterests(X,Y).
r10	hasAdvisor(X,Y) $\Rightarrow$ canRequestRecommendationLetter(X,Y).
r11	teaches(Y,X) $\Rightarrow$ canRequestRecommendationLetter(X,Y).
r12	teaches(Y,X), PostgraduateStudent(Y) $\Rightarrow \neg$ canRequestRecommendationLetter(X,Y).
	r12 > r11.
r13	Professor(X), worksFor(X,D), subOrganizationOf(D,U) $\Rightarrow$ canBecomeDean(X,U).
r14	Professor(X), headOf(X,D), subOrganizationOf(D,U) $\Rightarrow \neg$ canBecomeDean(X,U).
	r14 > r13.
r15	worksFor(X,D) $\Rightarrow$ canBecomeHeadOf(X,D).
r16	worksFor(X,D), headOf(Z,D), commonResearchInterests(X,Z) $\Rightarrow \neg$ canBecomeHeadOf(X,D).
	r16 > r15.
r17	teaches(Y,X) $\Rightarrow$ suggestAdvisor(X,Y).
r18	teaches(Y,X), hasAdvisor(X,Z) $\rightsquigarrow \neg$ suggestAdvisor(X,Y).
	r18 > r17.

performance systems, we have defined scalability as the ability to process datasets of increasing size in a proportional amount of time and the ability of our system to perform well as the computational resources increase. With regard to the former, we have performed experiments using datasets of various sizes (yet similar characteristics).

With regard to scaling computational resources, it has been empirically observed that the main inhibitor of parallel reasoning systems has been load-balancing between compute nodes [33]. Thus, we have also focused our scalability evaluation on this aspect.

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

**Platform.** Our experiments were performed on a IBM x3850 server with 40 cores and 750GB of RAM, connected to a XIV Storage Area Network (SAN), using a 10Gbps storage switch. We have used IBM Hadoop Cluster v1.3, which is compatible with Hadoop v0.20.2, along with an optimization to reduce Map task overhead, in line with the work of Vernica et al. [61]. Although our experiments were run on a single machine, there was no direct communication between processes and all data was transferred through persistent storage. We have used a number of Mappers and Reducers equal to the number of cores in the system (i.e. 40).

### 4.3.1 LUBM Use Case

**Dataset.** We have used the most popular benchmark for reasoning systems, LUBM<sup>2</sup>. LUBM allows us to scale the size of the data to an arbitrary size while keeping the reasoning complexity constant. For our experiments, we generated up to 8000 universities resulting in approximately 1 billion triples.

**Rule set.** The logic of LUBM can be partially expressed using RDFS and OWL2-RL. Nevertheless, neither of these logics are defeasible. Thus, to evaluate our system, we have created the ruleset in Table 4.1. The following predicates, used in our rule set, are found in LUBM ontology as owl classes: *FullProfessor*, *AssociateProfessor*, *AssistantProfessor*, *Professor* and *GraduateStudent* (used as: *PostgraduateStudent*). The following predicates, used in our rule set, are found in LUBM ontology as owl properties: *publicationAuthor*, *teacherOf*, *takesCourse*, *teachingAssistantOf*, *advisor* (used as: *hasAdvisor*), *researchInterest* (used as: *hasResearchInterest*), *worksFor*, *subOrganizationOf* and *headOf*. The remaining predicates have been introduced in our rule set in order to facilitate defeasible

---

<sup>2</sup><http://swat.cse.lehigh.edu/projects/lubm/>

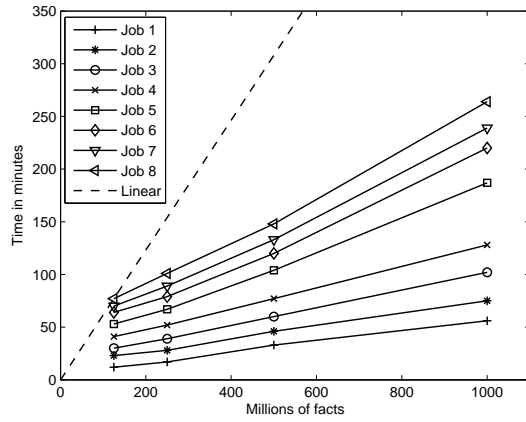


Figure 4.7: Runtime in minutes for various datasets, and projected linear scalability. Job runtimes are stacked (i.e. runtime for Job 8 includes the runtimes for Jobs 1-7).

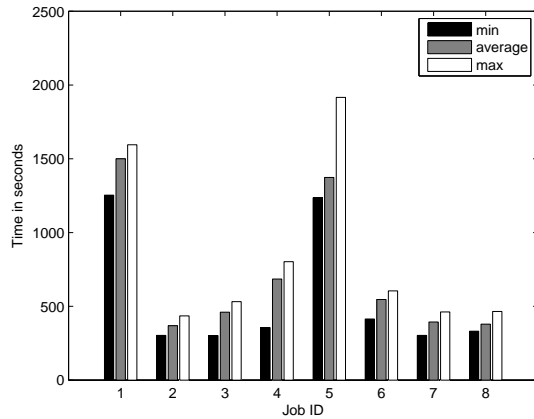


Figure 4.8: Minimum, average and maximum reduce task runtime for each job with 40 reduce tasks.

reasoning.

**MapReduce jobs description.** We need 8 jobs in order to perform reasoning on the above rule set. The *first* job is the *initial pass* described in Section 4.2.1 (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 *second* job computes rules r4-r6. During the *third* job we perform duplicate elimination, since r4-r6 are strict rules. We compute rules r7-r14 during the *fourth* job while reasoning on them, is performed during the *fifth* job. Jobs *six* and *seven* compute rules r15-r18. Finally, during the *eighth* job we perform reasoning on r15-r18, finishing the whole procedure.

**Results.** Figure 4.7 shows the runtimes of our system for varying input sizes. We make

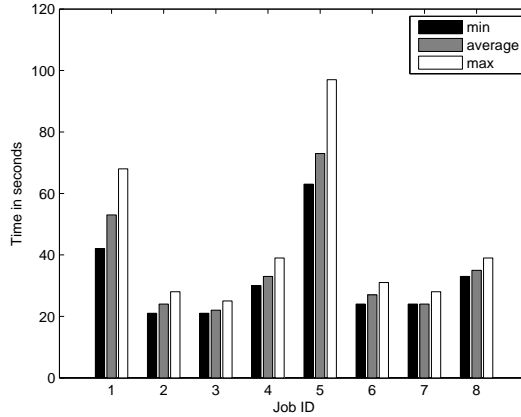


Figure 4.9: Minimum, average and maximum reduce task runtime for each job with 400 reduce tasks.

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

The above show that our system is indeed capable of achieving high performance and scales very well with the size of the input. Nevertheless, to further investigate how our system would perform when the data size precludes the use of a single machine, it is critical to examine the load-balancing properties of our algorithm.

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

Figures 4.8 and 4.9 show the load balance between different reduce tasks, for 1 billion triples and 40 (Figure 4.8) or 400 (Figure 4.9) reduce tasks. In principle, an application performs badly when a single task dominates the runtime, since all other tasks would need to wait for it to finish. In our experiments, it is evident that no such task exists. In addition, one may note that the system is actually faster with 400 reduce tasks. This is attributed both to the fact that each core in our platform can process two threads in parallel, and to implementation aspects of Hadoop that result in tasks, processing

approximately 1GB, demonstrating higher throughput than larger tasks.

Although a direct comparison is not meaningful, the throughput of our system is in line with results obtained when doing monotonic reasoning using state of the art RDF stores and inference engine. For example, OWLIM claims a 14.4-hour loading time for the same dataset when doing OWL horst inference <sup>3</sup>. WebPIE [47], which is also based on MapReduce, presents an OWL-horst inference time of 35 minutes, albeit on 64 lower-spec nodes and requiring an additional dictionary encoding step.

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

Finally, we should take into consideration the fact that LUBM produces fairly uniform data. Although there is significant skew in LUBM (e.g. in the frequency of terms such as `rdf:type`), the rule set that we have used in the evaluation does not perform joins on such highly skewed terms. However, our previous work [13] show that our approach can cope with highly skewed data, which follow a zipf distribution.

---

<sup>3</sup><http://www.ontotext.com/owlim/benchmark-results/lubm>



## Chapter 5

# Stratified Semantics of Logic Programs

In this chapter, a restricted version of the well-founded semantics is studied, namely the stratified semantics of logic programs, as it provides a more efficient computation for stratified programs.

### 5.1 Algorithm Description

In this subsection we present a parallel solution for stratified programs, address several special cases and discuss arising challenges.

According to the Definition 2.1.3, for a stratified program, literals that are given as input (facts) are assigned rank 0. Predicates that are supported by rules containing no negative subgoals are assigned rank 0, as well. Subsequently, literals depending negatively only on rank 0, are assigned rank 1. In general, literals depending negatively on rank  $k-1$ , are assigned rank  $k$ . Thus, stratification is finished when all predicates are assigned a rank.

Consider the following program:

$$\begin{aligned} r(X,Y) &\leftarrow q(X,Y), \text{ not } p(X,Y). \\ p(X,Y) &\leftarrow a(X,Z), b(Z,Y), \text{ not } c(X,Z), \text{ not } d(Y). \end{aligned}$$

Figure 5.1 shows how predicates are assigned to ranks, which in our example is as follows:

rank 0:  $a, b, c, d, q$   
rank 1:  $p$   
rank 2:  $r$

plain lines represent a positive dependency, while dashed lines represent a negative dependency.

Once predicates have been assigned to ranks, we proceed with reasoning according to the following algorithm:

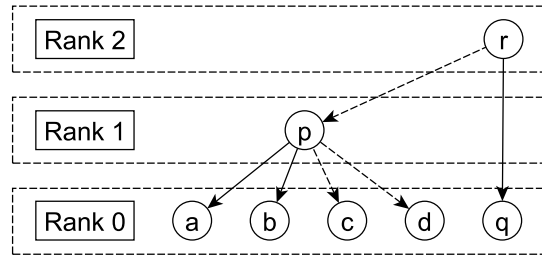


Figure 5.1: Predicates assigned to ranks.

Overall reasoning process:

```

Set KB = facts;
for (i=0; i<=N; i++) do
  do
    new_KB = Perform_reasoning(i, KB);
    KB += new_KB;
  while (new_KB != null) \\ check if fixpoint is reached
done

```

Initially, we add facts to our knowledge base (KB). Subsequently, for each rank (0 to  $N$ , where  $N$  is the highest rank for a given program) we perform reasoning as described later in this subsection (`Perform_reasoning(i, KB)`), by taking into consideration the rank we are reasoning for ( $i$ ) and the current knowledge base (KB), while classifying all the inferred literals as positive. Note that for each rank, reasoning is iterated until no new conclusion is derived (`while (new_KB != null)`). Once all inferences are computed for rank  $i$ , we continue with the next rank, until all ranks are evaluated. Finally, when reasoning is finished, literals that are contained in our KB are classified as positive, while those that are not are classified as negative.

Consider the following rule from the aforementioned program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(X,Z), \mathbf{not} d(Y).$$

here  $p(X,Y)$  is our *final goal*,  $a(X,Z)$  and  $b(Z,Y)$  are positive subgoals, and  $\mathbf{not} c(X,Z)$  and  $\mathbf{not} d(Y)$  are negative subgoals. We will group all positive subgoals into a *positive goal*. A positive goal consists of a new predicate (say  $ab$ ) that contains as arguments all the arguments of the final goal ( $X,Y$ ) plus all the common arguments with the negative subgoals ( $X,Z,Y$ ), namely we need to compute  $ab(X,Z,Y)$ . However, we need to ensure that for each rule, all arguments of the final goal ( $X,Y$ ) can be found in a positive subgoal. In addition, we need to ensure that for each rule, the set of arguments that can be found in all the negative subgoals ( $X,Z,Y$ ) is a subset ( $\subseteq$ ) of the arguments that can be found in all the positive subgoals ( $X,Z,Y$ ). In order to compute the final goal ( $p(X,Y)$ ) we retain

all values of the positive goal ( $ab(X,Z,Y)$ ) that have no equal values with any negative subgoal (**not**  $c(X,Z)$  and **not**  $d(Y)$ ) on their common arguments ( $X,Z$  and  $Y$  respectively).

As a general guideline, we perform a *single join* or *multiple joins* (see Subsection 5.1.1) in order to calculate positive goals, and *anti-joins* (see Subsection 5.1.2) to calculate the final goal. However, special cases may apply to certain programs (see Subsection 5.1.3).

### 5.1.1 Positive Goals Calculation

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y).$$

and the facts for  $a$  and  $b$

$$a(1,2) \quad a(1,3) \quad b(2,4) \quad b(3,4)$$

A single join can be performed either in *Map* or *Reduce*. However, the basic idea remains the same since in both cases we join  $a(X,Z)$  and  $b(Z,Y)$  on their common argument  $Z$  in order to produce  $p(X,Y)$ .

**Single join in Reduce.** We will first describe how joins (for the aforementioned rule) can be performed in *Reduce* according to the following pseudo-code:

```
map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (positive literal)
  if (value.predicate == "a") then
    emit(value.Z, {value.predicate,value.X});
  else if (value.predicate == "b") then
    emit(value.Z, {value.predicate,value.Y});

reduce(String key, Iterator values):
  // key: matching argument
  // values: positive literals for matching
  for each (value in values) do
    if (value.predicate == "a") then
      a_List.add(value.X)
    else
      b_List.add(value.Y)

  for each (a in a_List) do
    for each (b in b_List) do
      emit("p(a.X,b.Y)", "");
```

The *Map* function will emit pairs of the form  $\langle Z, (a, X) \rangle$  for predicate  $a$  and  $\langle Z, (b, Y) \rangle$  for predicate  $b$ , namely the following pairs:

$$\begin{aligned} &\langle 2, (a, 1) \rangle \\ &\langle 3, (a, 1) \rangle \\ &\langle 2, (b, 4) \rangle \\ &\langle 3, (b, 4) \rangle \end{aligned}$$

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

$$\begin{aligned} &\langle 2, \langle (a, 1), (b, 4) \rangle \rangle \\ &\langle 3, \langle (a, 1), (b, 4) \rangle \rangle \end{aligned}$$

During the reduce phase we match predicates  $a$  and  $b$  on their common argument (which is the *key*) and use the values to emit new literals. Thus, the reducer with key:

$$\begin{aligned} 2 &\text{ will emit } p(1, 4) \\ 3 &\text{ will emit } p(1, 4) \end{aligned}$$

As we see in our simple example,  $p(1, 4)$  is inferred twice. We need to filter out duplicates as soon as possible because they will produce unnecessary duplicates as well, affecting the overall performance. For brevity, we do not provide pseudo-code for duplicate elimination as it is straightforward for readers that are familiar with the MapReduce framework.

**Single join in Map.** In case of highly skewed data distribution, joins cannot be performed during *Reduce* due to skewed workload, which affects severely parallelization. However, such joins can be performed efficiently during *Map* if at least one of the two relations fits in main memory. As a real-world example, Kotoulas et al. [33] and Duan et al. [62] show that semantic web data are highly skewed following zipf distribution. Joins can be performed in *Map* according to the following pseudo-code:

```
// Create an in memory Map from a.Z to a.X
a_HashMap = load_literals_with_predicate_a();
map(Long key, String value):
  // key: position in document
  // value: document line (positive literal)
  if (value.predicate == "b") then
    if (a_HashMap.contains(b.Z)) then
      for each (X in a_HashMap.key(b.Z).iterator()) do
        emit("p(X,b.Y)", "");
```

First, we need to load in memory facts for predicate  $a$ , namely  $a(1, 2)$  and  $a(1, 3)$ , and create a HashMap from  $a.Z$  to  $a.X$  (`load_literals_with_predicate_a()`) in order to reassure quick lookups. During *Map*, we traverse through given data and for each literal with

predicate  $b$  (value.predicate == "b"), we lookup the HashMap (`a_HashMap.contains(b.Z)`) for matching  $Z$  values. In case the two relations can be joined on their common argument ( $Z$ ), we calculate and emit a new literal ( $p(X,b.Y)$ ) for each  $X$  in the HashMap. During the map phase the *Map* function with value:

$$\begin{aligned} b(2,4) & \text{ will emit } p(1,4) \\ b(3,4) & \text{ will emit } p(1,4) \end{aligned}$$

For joins that are performed in *Map*, we can use the reduce phase for duplicate elimination. Duplicates are grouped together during grouping/sorting. Thus, for each group (*Reduce* function) we eliminate duplicates by emitting each literal (*key*) only once (in this case *values* are ignored). For brevity, we do not provide pseudo-code for the *Reduce* as it is straightforward for readers that are familiar with the MapReduce framework.

**Multiple joins** We have already described how to perform a single join. However, we may need to perform multiple joins (multi-way join) in order to compute a positive goal. Consider the following program:

$$\begin{aligned} q(X,Y) & \leftarrow a(X,Z), b(Z,W), c(W,Y). \\ p(X,Y) & \leftarrow c(W,Y), b(Z,W), a(X,Z). \end{aligned}$$

In order to compute  $q(X,Y)$ , we can apply our approach for single join twice, by first joining  $a(X,Z)$  and  $b(Z,W)$  on  $Z$ , producing a temporary literal (say  $ab(X,W)$ ), and then join  $ab(X,W)$  and  $c(W,Y)$  on  $W$  (producing the final goal). However, we can optimize multi-way join by taking into consideration the whole program instead of computing each rule separately. Note that the body of both rules is practically identical. Both rules consist of the same three predicates ( $a,b,c$ ), which have the same common arguments ( $a,b$  have  $Z$  and  $b,c$  have  $W$ ). Thus, we may perform the required joins once and produce new literals for both  $q(X,Y)$  and  $p(X,Y)$ .

Multi-way join have been described in the work of Fische [63] and optimized by Afrati et al. [64]. In order to achieve an efficient implementation, optimizations in the work of Afrati et al. [64] should be taken into consideration. However, the proposed optimizations require better knowledge of the available data than our general assumptions on data distribution (uniform or skewed).

### 5.1.2 Final Goal Calculation

Once positive goals are calculated, we need to perform anti-joins with negative subgoals on their common arguments in order to retain as final goals literals whose values are found in the positive goal, but not in any of the negative subgoals. Consider the following rule:

$$a(X) \leftarrow b(X), \text{ not } c(X).$$

we need to retain values of  $X$  that are found in  $b$ , but not in  $c$ . In order to perform an anti-join, we follow the aforementioned approach for a single join, however, emitting a final goal for each value of  $X$  that is supported by the predicate  $b$ , and not by the predicate  $c$ .

### 5.1.3 Special Cases

**Goals with no common arguments.** Once we have calculated a positive goal, prior to performing anti-joins, we need to take into consideration if the positive goal and the negative subgoals have common arguments. Consider the following rule:

$$a(X) \leftarrow b(X), \mathbf{not} c(Y).$$

here  $b(X)$  is the positive goal and  $\mathbf{not} c(Y)$  is the negative subgoal. However, the positive goal and the negative subgoal have no common arguments. Thus, every positive goal will be included as final goal since there is no negative subgoal preventing conclusions. In this case we can optimize by omitting anti-joins since they do not affect the final result.

**Cartesian product.** The calculation of a positive goal may depend on positive subgoals that have no common arguments. Such calculation results in a cartesian product. Consider the following rule:

$$a(X,Y) \leftarrow b(X), c(Y), \mathbf{not} d(X).$$

In order to compute the positive goal (say  $bc(X,Y)$ ) we need to compute the cartesian product of  $b(X)$  and  $c(Y)$ . To the best of our knowledge there is no efficient solution proposed in the literature for cartesian product computation for the MapReduce framework. However, if one of  $b(X)$  or  $c(Y)$  fits in memory (say  $b(X)$ ), then cartesian product can be computed in the same fashion as a single join that is performed in *Map*. We load  $b(X)$  in main memory, while a *Map* function is applied on each  $c(Y)$ . Thus, cartesian product is produced by matching each  $c(Y)$  with every  $b(X)$  that is found in memory.

**Nested subgoals.** For simplicity of presentation we focused on rules where  $\mathbf{not}$  was applied only on literals. However, our approach can be generalized for programs containing rules where  $\mathbf{not}$  applies to a conjunction of literals. Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} (c(X,Z), \mathbf{not} d(Y)).$$

We can compute  $p(X,Y)$  by rewriting the program. We need to replace each  $\mathbf{not}$  that applies to a conjunction of literals with a logically equivalent expression by transforming the body of the rule into *Disjunctive Normal Form*. Let us perform the following transformations:

$$\begin{aligned} a(X,Z) \wedge b(Z,Y) \wedge \mathbf{not} (c(X,Z) \wedge \mathbf{not} d(Y)) &\equiv \\ a(X,Z) \wedge b(Z,Y) \wedge (\mathbf{not} c(X,Z) \vee d(Y)) &\equiv \\ (a(X,Z) \wedge b(Z,Y) \wedge \mathbf{not} c(X,Z)) \vee (a(X,Z) \wedge b(Z,Y) \wedge d(Y)) & \end{aligned}$$

Since we have a disjunction of conjunctive clauses, we may rewrite the program by replacing the aforementioned rule with a set of new rules. Specifically, for each conjunctive clause, we introduce a new rule, where the head of the new rule is the head of the initial rule ( $p(X,Y)$ ), while the body of the new rule is the conjunctive clause. Thus, for the aforementioned program the initial rule will be replaced by the following rules:

$$\begin{aligned} p(X,Y) &\leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(X,Z). \\ p(X,Y) &\leftarrow a(X,Z), b(Z,Y), d(Y). \end{aligned}$$

Once we have generated the new program, we may proceed with computing ranks and then perform reasoning as described above (provided that the new program is stratified and for each rule of the new program, all arguments of the final goal belong to a positive subgoal, while the set of arguments of negative subgoals is a subset ( $\subseteq$ ) of the set of arguments of positive subgoals).

#### 5.1.4 Final Remarks

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(X,Z), \mathbf{not} d(Y).$$

For simplicity of presentation we proposed the computation of the positive goal  $ab(X,Z,Y)$ , followed by two anti-joins, first with  $c(X,Z)$  and then with  $d(Y)$ . However, one can follow a more optimal approach by mixing the application of joins and anti-joins. Specifically, for the aforementioned program, we can perform an anti-join on  $a(X,Z)$  and  $c(X,Z)$ , producing  $ac(X,Z)$ , and an anti-join on  $b(Z,Y)$  and  $d(Y)$ , producing  $bd(Z,Y)$ . Subsequently, we join  $ac(X,Z)$  and  $bd(Z,Y)$  in order to compute the final goal.

The second approach is more optimal since it generates less intermediate results, while calculating the required final goal ( $p(X,Y)$ ). However, in order to reassure correct application of anti-joins, for each anti-join on a positive and a negative subgoal the following condition must hold: the set of arguments of the negative subgoal (NS) is a subset of the set of arguments of the positive subgoal (PS), while the two sets have at least one common argument, namely  $NS \subseteq PS$  and  $(NS \cap PS) \neq \emptyset$ .

Now let us point out the necessity of the imposed restrictions. It is required that for each rule, all arguments of the final goal belong to a positive subgoal. Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), \mathbf{not} b(Z,Y).$$

here for each value of  $X$  in  $a(X,Z)$ , we need to compute the following subset  $new\_Y = H\_U - Y\_In\_b(Z,Y)$ , where  $H\_U$  is the Herbrand universe and  $Y\_In\_b(Z,Y)$  is the set of values of  $Y$  that are found in  $b(Z,Y)$  such that  $a(X,Z)$  and  $b(Z,Y)$  have common values on  $Z$ . Such computation will introduce a significant overhead for the computation of  $new\_Y$ , and will require either a cartesian product of each value of  $X$  with its corresponding subset  $new\_Y$  (which is not applicable in general, see Subsection 5.1.3) or storing the final goal ( $p(X,Y)$ ) in the form  $p(X,new\_Y)$  which will result in long  $new\_Y$  sequences that will eventually affect parallelization.

We have posed an additional restriction, namely in order to perform an anti-join, the set of arguments of the negative subgoal must be a subset ( $\subseteq$ ) of the set of arguments of the positive subgoal. Consider the following program:

$$p(X,Y) \leftarrow a(X,Y), \mathbf{not} b(Y,Z).$$

here we cannot perform an anti-join on  $a(X,Y)$  and  $\mathbf{not} b(Y,Z)$  in order to compute the final goal  $(p(X,Y))$  since for a given value of  $Y$  we need to check whether all literals  $b(Y, H\_U)$ , where  $H\_U$  is the Herbrand universe, are classified as positive or negative. Thus, for given  $X, Y$  we may infer the final goal  $(p(X,Y))$ , if there is at least one  $b(Y,Z)$  that is classified as negative. However, for more complex rules such as:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(Z,W), \mathbf{not} d(W,U).$$

an efficient implementation for MapReduce is yet to be defined since for a given  $Z$  we need to find a combination of  $c(Z,W)$  and  $d(W,U)$  that are both classified as negative, while avoiding the full materialization of negative literals. Full materialization of both positive and negative literals (Herbrand base) may easily become prohibiting even for small datasets, and thus, is not applicable to big data.

Finally, the approach for non-stratified programs is different and comes with certain challenges. An extension of our approach would be the computation of the well-founded semantics. However, the definition of unfounded sets (see Definition 2.1.9) affects the scalability of the whole process. In order to conclude that  $p \in A$  we need to check every instantiated rule  $R$  of  $\mathbf{P}$  whose head is  $p$ . Such evaluation has to be conducted by a single node (since the MapReduce framework does not allow communication between nodes during map or reduce phase), causing either main memory insufficiency or skewed load balancing, decreasing the overall parallelization.

**Soundness and completeness.** The proposed method for stratified semantics of logic programs is sound and complete since reasoning process is in line with the imposed stratification. Indeed, facts are added directly to the knowledge base, while stratification provides a well-defined reasoning sequence from lower to higher ranks, provided that the reasoning process at each rank is sound and complete, which leads to all derived conclusions being sound since reasoning for rank  $k$  is based on a sound and complete knowledge base for ranks up to  $k - 1$ , and a complete derivation of all possible conclusions since rules for rank  $k$  are applied until no new knowledge is derived. At each rank, computing the positive goal first ensures that no information is lost as all possible joins are performed, while anti-joins model negation as failure since every predicate of a negative subgoal belongs to a lower rank (for which we have sound and complete knowledge).

## 5.2 Experimental Evaluation

**Methodology.** In order to evaluate our approach, we searched for proposed benchmarks in the literature. Liang et al. [65] evaluate the performance of several rule engines on data that fit in main memory. However, our approach is targeted on data that exceed the capacity of the main memory. Thus, in order to perform evaluation, we adjusted certain



parameters of the proposed methodology by Liang et al. [65]. We evaluate our approach considering *large join tests*, *default negation* and *datalog recursion*, while the rest of the proposed evaluation metrics of Liang et al. [65] are not applicable. Specifically, we do not perform evaluation for *indexing* since the MapReduce framework does not provide such an option. We have not yet developed a complete system that could perform reasoning based on our approach, thus, all optimizations and cost-based analysis were performed manually. Finally, Liang et al. [65] separate *loading* and *inference* time, focusing on inference time. However, such a separation is difficult for our approach since loading and inference time may overlap.

**Dataset.** Liang et al. [65] based their experiments on datasets that consisted of up to several millions facts. We aim to evaluate our approach for up to 1 billion facts. The main goal of our approach is to evaluate performance in terms of execution time and reassure scalability. Thus, we perform experiments for several dataset sizes for both *uniform* and *zipf* (highly skewed) data distributions.

**Large join tests.** Consider the following program:

$$p_i(X,Y) \leftarrow a(X,Z), b(Z,Y).$$

for  $1 \leq i \leq N$ , where  $N$  is the total number of rules following the above rule pattern. Scalability of large joins is examined over several dataset sizes, number of rules, data distributions and number of nodes.

**Default negation.** Here we provide a solution for stratified programs. Thus, we cannot use the well known win-not-win example presented by Gelder [24] (also used in Liang et al. [65]), which works for locally stratified and non-stratified programs. Consider the following program:

$$p(X,Y) \leftarrow a(X,Y), \mathbf{not} \ b(X,Y).$$

Scalability of anti-joins is examined over several dataset sizes, data distributions and number of nodes. The number of rules has the same effect for anti-joins as for joins (since it affects only the total amount of final output), and thus, such an evaluation is omitted.

**Datalog recursion.** In order to evaluate datalog recursion, we applied a different evaluation method. Instead of generating random data and calculating the transitive closure of a relation, we evaluated joins using the program described in *large join tests* (for  $N = 1$ ). We performed joins, for uniform data distribution (zipf distribution would require more complex computation techniques, which are out of the scope of this work), changing the percentage of the matched values for the argument  $Z$  from 0% to 100%. In such a way, we were able to estimate the required time for each matching percentage, as this percentage may vary significantly throughout the transitive closure calculation.

**Platform.** We have implemented our experiments using the Hadoop MapReduce framework<sup>1</sup>, version 1.0.4. We have performed experiments on a cluster of the University

<sup>1</sup><http://hadoop.apache.org/mapreduce/>

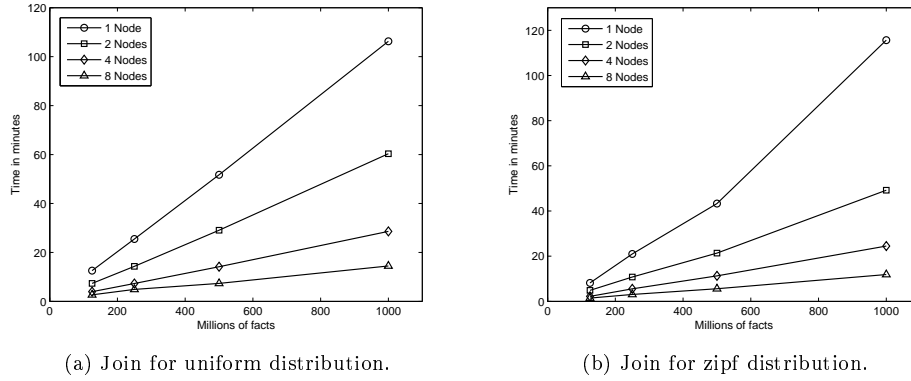


Figure 5.2: Runtime in minutes for join operations as a function of dataset size, for various numbers of nodes.

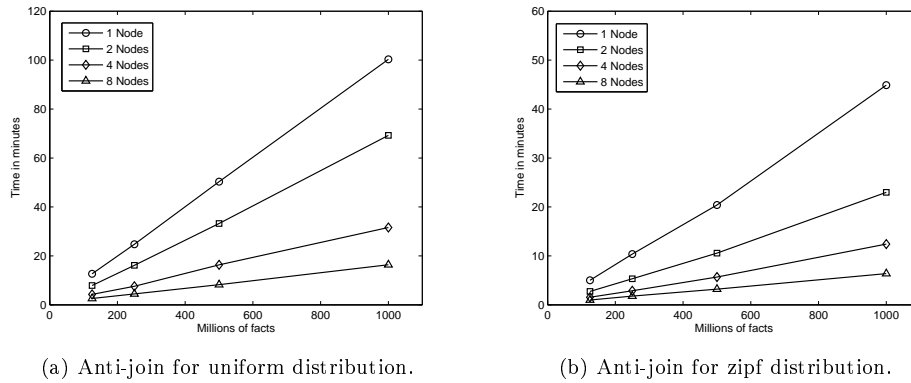


Figure 5.3: Runtime in minutes for anti-join operations as a function of dataset size, for various numbers of nodes.

of Huddersfield. The cluster consists of 9 nodes (one node was allocated as “master” node), using a Gigabit Ethernet interconnect. Each node was equipped with 2 cores running at 1.86GHz, 3GB RAM and 150GB of storage space.

**Results.** We can identify four main factors that affect the performance of our approach:

1. **Number of facts**, affecting the input size.
2. **Number of rules**, affecting the output size.
3. **(Anti-)Join percentage**, affecting the output size.
4. **Compression ratio**, affecting the output size, when compression algorithm is used.

our results correspond to several combinations of the above four factors.

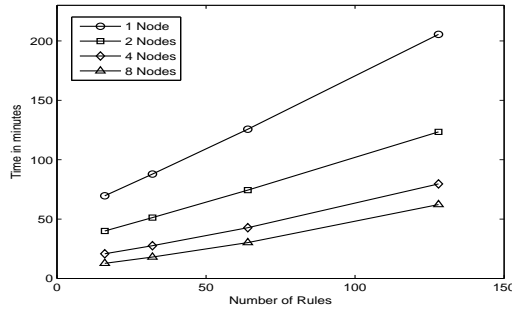


Figure 5.4: Runtime in minutes for various numbers of rules and nodes.

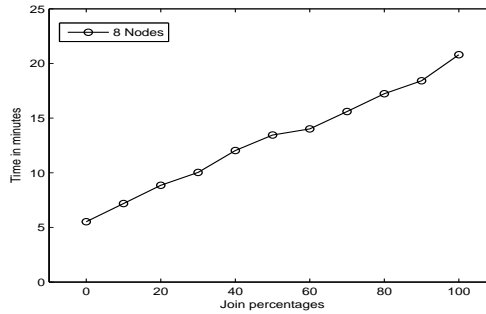


Figure 5.5: Runtime in minutes for various matched values percentages.

Figure 5.2 shows the runtimes of our system for join operations with input sizes up to 1 billion facts, while number of rules is set to 1 and compression ratio remains fairly stable. We see that for uniform distribution our approach scales linearly, as join percentage remains stable at 50%. However, for zipf distribution our approach exhibits several fluctuations as the number of facts increases, since the join percentage fluctuates as well.

Figure 5.3 presents the runtimes of our system for anti-join operations with input sizes up to 1 billion facts. For both uniform and zipf distribution, the number of rules is set to 1, while compression ratio and anti-join percentages remain stable. In this case we see that our system scales linearly for both data distributions.

Figure 5.4 depicts the scaling properties of our system for 16, 32, 64 and 128 rules. We need to point out that the system scales linearly for up to 64 rules, while for 128 rules the runtime is higher than the expected (linear). This is attributed to the fact that compression for 128 rules is less effective, resulting in larger amounts of output.

Figure 5.5 illustrates the runtimes of our system for various join percentages, while all the other factors remain stable (500 million facts, 1 rule, 8 nodes and fairly stable compression ratio). As expected, while the join percentage increases, the runtime increases as well since larger amounts of output are being produced. In general, for the case of recursion, long chains of MapReduce jobs and low join percentages should be avoided

because reading and sorting/grouping the input introduces a high overhead for the whole computation.

# Chapter 6

## Well-Founded Semantics

In this chapter, an approach for the full well-founded semantics (including the subset of stratified programs) is presented, which however is computationally more expensive compared to the approach for the stratified semantics of logic programs. In addition, we provide an analysis comparing both methods, thus showing the computational advantages of the stratified over the full well-founded semantics approach for the restricted subset of stratified programs.

### 6.1 Full Well-Founded Semantics

#### 6.1.1 Join and Anti-join for WFS

In this subsection we provide a description of the  $T_{P,J}(I)$  computation, which is modeled as a sequence of join and anti-join operations.

##### Computing $T_{P,J}(I)$

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(X,Z), \mathbf{not} d(Z,Y).$$

here  $p(X,Y)$  is our *final goal*,  $a(X,Z)$  and  $b(Z,Y)$  are positive subgoals, while  $\mathbf{not} c(X,Z)$  and  $\mathbf{not} d(Z,Y)$  are negative subgoals. In order to compute our final goal  $p(X,Y)$  we need to ensure that  $\{a(X,Z), b(Z,Y)\} \subseteq I$  and  $\{c(X,Z), d(Z,Y)\} \cap J = \emptyset$ , namely both  $a(X,Z)$  and  $b(Z,Y)$  are in  $I$  while none of  $c(X,Z)$  and  $d(Z,Y)$  is found in  $J$ .

As positive subgoals depend on  $I$  we can group them into a *positive goal*. A positive goal consists of a new predicate (say  $ab$ ) that contains as arguments the union of two sets: (a) all the arguments of the final goal  $(X,Y)$  and (b) all the common arguments between positive and negative subgoals  $(X,Z,Y)$ , namely we need to compute  $ab(X,Z,Y)$ . The final goal  $(p(X,Y))$  consists of all values of the positive goal  $(ab(X,Z,Y))$  that do not match any

**Algorithm 4** Single join

---

```

map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (literal in I)
  if (value.predicate == "a") then
    emit(value.Z, {value.predicate,value.X});
  else if (value.predicate == "b") then
    emit(value.Z, {value.predicate,value.Y});

reduce(String key, Iterator values):
  // key: matching argument
  // values: literals in I for matching
  a_List.empty();
  b_List.empty();
  for each (value in values) do
    if (value.predicate == "a") then
      a_List.add(value.X)
    else
      b_List.add(value.Y)
  for each (a in a_List) do
    for each (b in b_List) do
      emit("ab(a.X,key.Z,b.Y)","");

```

---

of the negative subgoals (**not**  $c(X,Z)$  and **not**  $d(Z,Y)$ ) on their common arguments ( $X,Z$  and  $Z,Y$  respectively).

**Positive Goal Calculation**

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \text{not } c(X,Z), \text{not } d(Z,Y).$$

where

$$\begin{aligned}
 I &= \{a(1,2), a(1,3), b(2,4), b(3,5)\} \\
 J &= \{c(1,2), d(2,3)\}
 \end{aligned}$$

A single join, calculating the positive goal  $ab(X,Z,Y)$ , can be performed as described in Algorithm 4. Note that we use only literals from  $I$ .

The *Map* function will emit pairs of the form  $\langle Z, (a,X) \rangle$  for predicate  $a$  and  $\langle Z, (b,Y) \rangle$  for predicate  $b$ , namely the following pairs:

$$\begin{aligned}
 &\langle 2, (a,1) \rangle \\
 &\langle 3, (a,1) \rangle \\
 &\langle 2, (b,4) \rangle \\
 &\langle 3, (b,5) \rangle
 \end{aligned}$$

---

**Algorithm 5** Duplicate elimination

---

```

map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (inferred literal)
  emit(value, "");

reduce(String key, Iterator values):
  // key: inferred literal
  // values: empty values (not used)
  emit(key, "");

```

---

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

$$\begin{aligned} &\langle 2, \langle (a,1), (b,4) \rangle \rangle \\ &\langle 3, \langle (a,1), (b,5) \rangle \rangle \end{aligned}$$

During the reduce phase we match predicates  $a$  and  $b$  on their common argument (which is the *key*) and use the values to emit positive goals. Thus, the reducer with key:

$$\begin{aligned} 2 &\text{ will emit } ab(1,2,4) \\ 3 &\text{ will emit } ab(1,3,5) \end{aligned}$$

Note that we need to filter out possibly occurring duplicates as soon as possible because they will produce unnecessary duplicates as well, affecting the overall performance. Pseudo-code for duplicate elimination is depicted in Algorithm 5.

Practically, the *Map* function emits every inferred literal as the key, with an empty value. The MapReduce framework performs grouping/sorting resulting in one group (of duplicates) for each unique literal. Each group of duplicates consists of the unique literal as the key and a set of empty values (with values being eventually ignored). The actual duplicate elimination takes place during the reduce phase since for each group of duplicates, we emit the (unique) inferred literal once, using the key, while ignoring the values.

For rules with more than one join between positive subgoals we need to apply multi-joins (multi-way join).

Consider the following program:

$$q(X,Y) \leftarrow a(X,Z), b(Z,W), c(W,Y), \mathbf{not} d(X,W).$$

We can compute the positive goal ( $abc(X,W,Y)$ ) by applying our approach for single join twice. First, we need to join  $a(X,Z)$  and  $b(Z,W)$  on  $Z$ , producing a temporary literal (say  $ab(X,W)$ ), and then join  $ab(X,W)$  and  $c(W,Y)$  on  $W$  producing the positive goal ( $abc(X,W,Y)$ ). Once  $abc(X,W,Y)$  is calculated, we proceed with calculating the final goal  $q(X,Y)$  by retaining all the values of  $abc(X,W,Y)$  that do not match  $\mathbf{not} d(X,W)$  on their common arguments  $(X,W)$ .

**Algorithm 6** Anti-join

---

```

map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (literal)
  if (value.predicate == "ab") then
    emit({value.X,value.Z}, {value.predicate,value.Y});
  else if (value.predicate == "c") then
    emit({value.X,value.Z}, value.predicate);

reduce(String key, Iterator values):
  // key: matching argument
  // values: literals for matching
  ab_List.empty();
  for each (value in values) do
    if (value.predicate == "ab") then
      ab_List.add(value.Y)
    else
      return; // matched by predicate c
  for each (ab in ab_List) do
    emit("abc(key.X,key.Z,ab.Y)", "");

```

---

For details on single and multi-way join, readers are referred to literature. More specifically, multi-way join has been described and optimized by Afrati et al. [64]. In order to achieve an efficient implementation, the proposed optimizations by Afrati et al. [64] should be taken into consideration.

**Final Goal Calculation**

Consider the aforementioned program:

$$p(X,Y) \leftarrow a(X,Z), b(Z,Y), \mathbf{not} c(X,Z), \mathbf{not} d(Z,Y).$$

where

$$\begin{aligned}
I &= \{a(1,2), a(1,3), b(2,4), b(3,5)\} \\
J &= \{c(1,2), d(2,3)\}
\end{aligned}$$

By calculating the positive goal  $ab(X,Z,Y)$  we obtain the following knowledge:

$$\begin{aligned}
&ab(1,2,4) \\
&ab(1,3,5)
\end{aligned}$$

In order to calculate the final goal ( $p(X,Y)$ ) we need to perform an anti-join between  $ab(X,Z,Y)$  and each negative subgoal ( $\mathbf{not} c(X,Z)$  and  $\mathbf{not} d(Z,Y)$ ). Note that to perform an anti-join we use only the previously calculated positive goal ( $ab(X,Z,Y)$ ) and literals from  $J$ .



We start by performing an anti-join between  $ab(X,Z,Y)$  and **not**  $c(X,Z)$  on their common arguments  $(X,Z)$ , creating a new literal (say  $abc(X,Z,Y)$ ), which contains all the results from  $ab(X,Z,Y)$  that are not found in  $c(X,Z)$ , as described in Algorithm 6.

The *Map* function will emit pairs of the form  $\langle(X,Z),(ab,Y)\rangle$  for predicate  $ab$  and  $\langle(X,Z),c\rangle$  for predicate  $c$  (while predicate  $d$  will be taken into consideration during the next anti-join), namely the following pairs:

$$\begin{aligned} &\langle(1,2), (ab,4)\rangle \\ &\langle(1,3), (ab,5)\rangle \\ &\langle(1,2), c\rangle \end{aligned}$$

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

$$\begin{aligned} &\langle(1,2), \langle(ab,4), (c)\rangle\rangle \\ &\langle(1,3), (ab,5)\rangle \end{aligned}$$

During the reduce phase we output values of the predicate  $ab$  only if it is not matched by predicate  $c$  on their common arguments (which are contained in the *key*) and emit  $abc(X,Z,Y)$ . Thus, the reducer with key:

$$\begin{aligned} (1,2) &\text{ will have no output} \\ (1,3) &\text{ will emit } abc(1,3,5) \end{aligned}$$

In order to calculate the final goal  $(p(X,Y))$ , we need to perform an additional anti-join between  $abc(X,Z,Y)$  and  $d(Z,Y)$  on their common arguments  $(Z,Y)$ . Here,  $abc(1,3,5)$  and  $d(2,3)$  do not match on their common arguments  $(Z,Y)$  as  $(3,5) \neq (2,3)$ . Thus, our calculated final goal is  $p(1,5)$ .

### 6.1.2 Computing the Well-Founded Semantics

In this subsection we describe a naive and an optimized implementation for the calculation of the well-founded semantics, and provide a proof sketch in order to justify the correctness of our approach.

#### Naive Implementation

A naive implementation of the well-founded semantics fixpoint is depicted in Algorithm 7. The algorithm takes as input a program  $P$  and calculates the sets of literals  $K_i$  and  $U_i$  until fixpoint is reached, namely  $(K_{i-1}, U_{i-1}) = (K_i, U_i)$  for  $i \geq 1$ . Each set of literals  $(K_i$  and  $U_i)$  is calculated by the naive least fixpoint of  $T_{P,J}(I)$  depicted in Algorithm 8.

The naive least fixpoint of  $T_{P,J}(I)$  (*naive\_lfp*) takes as input two arguments, a program  $P$  and a set of literals  $J$ . Practically, we calculate the least fixpoint of  $T_{P,J}(I)$  where  $P$  and  $J$  are given as input while  $I$  is initially set as empty ( $I = \emptyset$ ). We also use a temporary set of inferred literals (*new*) in order to eliminate duplicates ( $new = new - I$ ) prior to

**Algorithm 7** Naive WFS fixpoint

---

```

naive_WFS_fixpoint(P):
  // input: program P,
  // output: set of literals  $K_i, U_i$ 
   $K_0 = \text{naive\_lfp}(P+, \emptyset)$ ;
   $U_0 = \text{naive\_lfp}(P, K_0)$ ;
   $i = 0$ ;
  do
     $i++$ ; // next "inference step"
     $K_i = \text{naive\_lfp}(P, U_{i-1})$ ;
     $U_i = \text{naive\_lfp}(P, K_i)$ ;
  while ( $K_{i-1} \neq K_i$  or  $U_{i-1} \neq U_i$ )
  return  $K_i, U_i$ ;

```

---

**Algorithm 8** Naive least fixpoint of  $T_{P,J}(I)$ 


---

```

naive_lfp(P, J):
  // input: program P, set of literals J
  // output: set of literals I (least fixpoint of  $T_{P,J}(\emptyset)$ )
   $I = \emptyset$ ;
   $new = \emptyset$ ;
  do
     $I = I \cup new$ ;
     $new = T(P, I, J)$ ;
     $new = new - I$ ;
  while ( $new \neq \emptyset$ )
  return I;

```

---

adding newly inferred literals to the set  $I$  ( $I = I \cup new$ ). Thus, we start by having  $I = \emptyset$  and stop when no new knowledge can be inferred ( $new = \emptyset$ ). The function  $T(P, I, J)$  is the computation of  $T_{P,J}(I)$  as described above.

Let us now consider the calculation of the naive WFS fixpoint. Initially, we calculate  $K_0$  over the positive part of the program  $P$  ( $P+$ ) and set  $J = \emptyset$ . Then, we proceed with the calculation of  $U_0$  given the already available set  $K_0$  ( $J = K_0$ ). Subsequently, we increase the counter  $i$  and calculate the least fixpoint of  $K_i$  (resp.  $U_i$ ) given  $U_{i-1}$  (resp.  $K_i$ ) until fixpoint is reached. WFS fixpoint is reached when  $K_{i-1} = K_i$  and  $U_{i-1} = U_i$ , and finally the sets of literals  $K_i$  and  $U_i$  are returned.

According to Theorem 2.1.2, having reached WFS fixpoint at step  $i$ , we can determine which literals are true, undefined and false as follows:

- **true** literals, denoted by  $K_i$ .
- **undefined** literals, denoted by  $U_i - K_i$ .

- **false** literals, denoted by  $\text{BASE}(P) - U_i$ .

A straightforward optimization of the naive WFS fixpoint algorithm is to store the sets of literals  $K_i$  and  $U_i$  only for the current and the previous “inference step”, namely if  $i = k$ , for  $k \geq 1$ , then we only need to store  $K_{i-1}$  and  $U_{i-1}$  in our knowledge base while calculating  $K_i$  and  $U_i$ . Since a fixpoint was not reached for  $i = k - 1$ , any  $K_j$  and  $U_j$  for  $j < k - 1$  becomes irrelevant as it will not be used for the remainder of the computation. Thus, at any time of calculation (step  $i$ , for  $i \geq 1$ ) we need to store up to four sets of literals ( $K_{i-1}$ ,  $U_{i-1}$ ,  $K_i$ ,  $U_i$ ).

### Optimized Implementation

The naive implementation, as presented above, introduces unnecessary overhead to the overall computation. A more refined version of both WFS fixpoint and least fixpoint of  $T_{P,J}(I)$  is defined in Algorithm 9 and Algorithm 10 respectively.

Our first optimization is the changed calculation of the least fixpoint of  $T_{P,J}(I)$  (*opt\_lfp*), which is depicted in Algorithm 10. Instead of calculating the least fixpoint starting from  $I = \emptyset$ , for a given program  $P$  and a set of literals  $J$ , as in Algorithm 8, we allow the calculation to start from a given  $I$ , provided that  $I \subseteq \text{lfp}(T_{P,J}(\emptyset))$ , and return only the newly inferred literals ( $S$ ) that led us to the least fixpoint. Thus, the actual set of literals that the least fixpoint of  $T_{P,J}(I)$  consists of is  $I \cup S$ . In order to reassure correctness we need to take into consideration both  $I$  and  $S$  while calculating the least fixpoint, namely new literals are inferred by calculating  $T_{P,J}(I \cup S)$ . However, as before, we use a temporary set of inferred literals (*new*) in order to eliminate duplicates ( $\text{new} = \text{new} - (I \cup S)$ ) prior to adding newly inferred literals to the set  $S$  ( $S = S \cup \text{new}$ ). Note that the set of literals  $I$  remains unchanged when the optimized least fixpoint is calculated.

The optimized version of the least fixpoint is used, in Algorithm 9, for the computation of each set of literals  $K$  and  $U$ .  $K_0$  is a special case where we start from  $I = \emptyset$  and  $J = \emptyset$ , and thus, unable to fully utilize the advantages of the optimized least fixpoint.

The proposed optimizations are mainly based on the monotonicity of the well-founded semantics as given in Lemma 2.1.1. Note that in this subsection, the indices of the sets  $K$  and  $U$  found in Lemma 2.1.1 are adjusted to the indices used in Algorithm 9 in order to facilitate our discussion.

Since  $K_i \subseteq U_i$ , for  $i \geq 0$  (see Lemma 2.1.1), the computation of  $U_i$  can start from  $K_i$ , namely  $I = K_i$ . Thus, instead of recomputing all literals of  $K_i$  while calculating  $U_i$ , we can use them to speed up the process. Note that the actual least fixpoint of  $U_i$  is the union of sets  $K_i$  and  $\text{opt\_lfp}(P, K_i, K_i)$ , as the optimized least fixpoint computes only new literals (which are not included in given  $I$ ).

Since  $K_{i-1} \subseteq K_i$ , for  $i \geq 1$  (see Lemma 2.1.1), the computation of  $K_i$  can start from  $K_{i-1}$ , namely  $I = K_{i-1}$ . Once  $\text{opt\_lfp}(P, K_{i-1}, U_{i-1})$  is computed, we append it to our previously stored knowledge  $K_{i-1}$ , resulting in  $K_i$ .

**Algorithm 9** Optimized WFS fixpoint

---

```

opt_WFS_fixpoint(P):
  // input: program P,
  // output: set of literals  $K_{i-1}, U_{i-1}$ 
   $K_0 = \text{opt\_lfp}(P, \emptyset, \emptyset)$ ;
   $i = 0$ ;
  do
     $U_i = K_i \cup \text{opt\_lfp}(P, K_i, K_i)$ ;
     $i++$ ; // next "inference step"
     $K_i = K_{i-1} \cup \text{opt\_lfp}(P, K_{i-1}, U_{i-1})$ ;
  while ( $K_{i-1}.\text{size}() \neq K_i.\text{size}()$ )
  return  $K_{i-1}, U_{i-1}$ ;

```

---

**Algorithm 10** Optimized least fixpoint of  $T_{P,J}(I)$ 


---

```

opt_lfp(P, I, J):
  // precondition:  $I \subseteq \text{lfp}(T_{P,J}(\emptyset))$ 
  // input: program P, set of literals I, set of literals J
  // output: set of literals S ( $\text{lfp}(T_{P,J}(I)) - I$ )
   $S = \emptyset$ ;
  new =  $\emptyset$ ;
  do
     $S = S \cup \text{new}$ ;
    new =  $T(P, (I \cup S), J)$ ;
    new = new -  $(I \cup S)$ ;
  while (new  $\neq \emptyset$ )
  return S;

```

---

**Lemma 6.1.1** *WFS fixpoint is reached at step  $i$ , for  $i \geq 1$ , if  $K_{i-1} = K_i$ .*

**Proof.** *If  $K_{i-1} = K_i$ , for  $i \geq 1$ , then*

$$\begin{aligned}
 U_{i-1} &= K_{i-1} \cup \text{opt\_lfp}(P, K_{i-1}, K_{i-1}) \\
 &= K_i \cup \text{opt\_lfp}(P, K_i, K_i) \\
 &= U_i
 \end{aligned}$$

*Thus, fixpoint is reached as  $(K_{i-1}, U_{i-1}) = (K_i, U_i)$ .*

Although for  $K_i$  calculation only new literals are inferred during each "inference step", for  $U_i$  we have to recalculate a subset of literals that can be found in  $U_{i-1}$ , as literals in  $U_{i-1} - K_{i-1}$  are discarded prior to the computation of  $U_i$ . However, the computational overhead coming from the calculation of  $\text{opt\_lfp}(P, K_i, K_i)$  reduces over time since the set of literals in  $U_i - K_i$  becomes smaller after each "inference step" due to  $K_{i-1} \subseteq K_i$  and  $U_{i-1} \supseteq U_i$ , for  $i \geq 1$ , (see Lemma 2.1.1).

We may further optimize our approach by minimizing the amount of stored literals. As

discussed in subsection 6.1.2, the naive implementation requires the storage of up to four overlapping sets of literals ( $K_{i-1}$ ,  $U_{i-1}$ ,  $K_i$ ,  $U_i$ ). However, as  $K_i \subseteq U_i$ , while calculating  $U_i$ , we need to store in our knowledge base only the sets  $K_i$  and  $opt\_lfp(P, K_i, K_i)$ , since  $U_i = K_i \cup opt\_lfp(P, K_i, K_i)$ .

As  $K_{i-1} \subseteq K_i$ , for the calculation of  $K_i$ , we need to store in our knowledge base only three sets of literals, namely: (a)  $K_{i-1}$ , (b)  $U_{i-1} - K_{i-1} = opt\_lfp(P, K_{i-1}, K_{i-1})$  and (c) currently calculating least fixpoint  $opt\_lfp(P, K_{i-1}, U_{i-1})$ . All newly inferred literals in  $opt\_lfp(P, K_{i-1}, U_{i-1})$ , are added to  $K_i$  (replacing our prior knowledge about  $K_{i-1}$ ), while literals in  $U_{i-1} - K_{i-1} = opt\_lfp(P, K_{i-1}, K_{i-1})$  are deleted, if fixpoint is not reached, as they cannot be used for the computation of  $U_i$ .

A WFS fixpoint is reached when  $K_{i-1} = K_i$ , namely when no new literals are derived during the calculation of  $K_i$ , which practically is the calculation of  $opt\_lfp(P, K_{i-1}, U_{i-1})$ . Since  $(K_{i-1}, U_{i-1}) = (K_i, U_i)$ , we return the sets of literals  $K_{i-1}$  and  $U_{i-1}$ , representing our fixpoint knowledge base.

In practice, the maximum amount of stored data occurs while calculating  $K_i$ , for  $i \geq 1$ , where we need to store three sets of literals, namely: (a)  $K_{i-1}$ , (b)  $U_{i-1} - K_{i-1}$  and (c)  $opt\_lfp(P, K_{i-1}, U_{i-1})$ , requiring significantly less storage space compared to the naive implementation.

### Approach Correctness - Soundness and Completeness

We need to ensure that our approach is in line with the definition of the alternating fixpoint procedure (see Definition 2.1.12) for both naive and optimized algorithms (see Algorithms 7, 8, 9, 10). By justifying the correctness of our approach we also ensure soundness and completeness, which is based on the soundness and completeness of the alternating fixpoint procedure. Thus, we provide the following proof sketch.

*Proof sketch.* First, we need to ensure that we calculate  $T_{P,J}(I)$  according to Definition 2.1.11. Consider a given program  $P$  and given sets of literals  $I$  and  $J$ .

According to subsection 6.1.1, the positive part ( $pos(\mathcal{B})$ ) of each rule of the instantiated program  $P$  ( $A \leftarrow \mathcal{B} \in ground(P)$ ) is calculated using literals from  $I$  ( $pos(\mathcal{B}) \subseteq I$ ), which agrees with Definition 2.1.11. Single and multi-way joins that are described in this work have been studied in the literature by Afrati et al. [64], and thus, ensure correct computation. The auxiliary predicates used in positive goal computation do not affect the final result as these predicates are not part of the given program. In addition, the fact that the positive goal contains the smallest set of arguments containing arguments of both final goal and negative subgoals reassures correctness, since no information is lost, while it minimizes the computation cost, as the overhead coming from redundant arguments is eliminated.

Since the positive goal is equivalent to the computation of the positive part of a rule, we may proceed with the negative part. According to Definition 2.1.11, a final goal of a rule is

computed from a set of positive subgoals that belong to  $I$  ( $A \mid \text{there is } A \leftarrow \mathcal{B} \in \text{ground}(P)$  with  $\text{pos}(\mathcal{B}) \subseteq I$ ), namely the positive goal of the rule, and a set of negative subgoals that do not belong to  $J$  ( $\text{neg}(\mathcal{B}) \cap J = \emptyset$ ). Thus, negative subgoals that could possibly match the positive goal on their common arguments should not be found in  $J$ . This is modeled by the anti-join as described in subsection 6.1.1.

Our next step is to investigate the equivalence of the least fixpoint calculation. According to the definition of the least fixpoint, provided in subsection 2.1.9, for a given program  $P$  and a set of literals  $J$  we start with  $I = \emptyset$  and gradually calculate applicable rules until no new literals are inferred, namely  $T_{P,J}(I) = I$ . This is directly modeled by the naive least fixpoint (see Algorithm 8), since the computation starts with an empty set ( $\emptyset$ ) and  $T_{P,J}(I)$  is applied until no new knowledge is derived. The case of the optimized least fixpoint (see Algorithm 10) is more complex despite the fact that for  $I = \emptyset$ , naive and optimized least fixpoint calculations are equivalent.

We need to point out that the optimized least fixpoint can only be applied when the given  $I$  is a subset ( $\subseteq$ ) of the calculation of the least fixpoint of  $T_{P,J}(\emptyset)$ . Indeed, if the least fixpoint of  $T_{P,J}(\emptyset)$  is the set of literals  $W$ , then only subsets of  $W$  can be given as  $I$ . If a given  $I$  is not a subset of  $W$  then it contains a literal (say  $p$  where  $p \notin W$ ) that will be included in the least fixpoint of  $T_{P,J}(I)$ , resulting in a set of literals  $V$  where  $V \neq W$ , and thus, leading to inconsistency. However, when the optimized least fixpoint calculation is applied starting from a given  $I$  where  $I \subseteq W$ , then it will eventually lead to the computation of  $W$ , while allowing a speed up of the computation process. In Algorithm 10, given a program  $P$  and sets of literals  $I$  and  $J$ , provided that  $I \subseteq W$ , the optimized least fixpoint calculates  $S = W - I$ , since  $I$  is already available in the knowledge base.

We have demonstrated that the calculation of the naive and the optimized least fixpoint is in line with the calculation of  $\text{lfp}(T_{P,J}(I))$  of the alternating fixpoint procedure (see subsection 2.1.9). The correctness of Algorithm 7 and Algorithm 9 is ensured by the carefully assigned sets of literals  $I$  and  $J$ , given a program  $P$ , for each  $\text{lfp}(T_{P,J}(I))$  taking into consideration the monotonicity of the alternating fixpoint procedure as described in subsection 6.1.2 and subsection 6.1.2.

### Computational Impact of Safety

Apart from the semantic motivation of the safety requirement outlined in subsection 2.1.8, it also has considerable impact on the computational method followed in this work. Recall that safety requires that each variable in a rule must occur (also) in a positive subgoal. If this safety condition is not met, an anti-join is no longer a single lookup between the positive goal and a negative subgoal, but a comparison between a subset of the Herbrand base and a given set of literals  $J$ . An efficient implementation for such computation is yet to be defined and problematic, as illustrated next.

Consider the following program:

$$\begin{aligned} p(X,Y) &\leftarrow a(X,Y), \mathbf{not} b(Y,Z). \\ q(X,Y) &\leftarrow c(X,U), \mathbf{not} d(W,U), \mathbf{not} e(U,Y). \end{aligned}$$

For the first rule, each  $(X, Y)$  in  $a(X, Y)$  is included in the final goal ( $p(X, Y)$ ) only if for a given  $Y$ , there is a  $Z$  in Herbrand universe such that  $b(Y, Z)$  does not belong to  $J$ . For the second rule, for each  $(X, Y)$  that is included in the final goal ( $q(X, Y)$ ) there should be a literal  $c(X, U)$  that does not match neither  $d(W, U)$  on  $U$ , for any  $W$  in Herbrand universe, nor  $e(U, Y)$  on  $U$ , for any  $Y$  in Herbrand universe. Thus, we need to perform reasoning over a subset of the Herbrand base for  $b(Y, Z)$ ,  $d(W, U)$  and  $e(U, Y)$  in order to find the nonmatching literals.

In this work, we follow the alternating fixpoint procedure in order to avoid full materialization of or reasoning over the Herbrand base for any predicate. Storing or performing reasoning over the entire Herbrand base may easily become prohibiting even for small datasets, and thus, not applicable to big data.

### 6.1.3 Experimental Results

**Methodology.** In order to evaluate our approach, we surveyed available benchmarks in the literature. Liang et al. [65] evaluate the performance of several rule engines on data that fit in main memory. However, our approach is targeted on data that exceed the capacity of the main memory. Thus, we follow the proposed methodology of Liang et al. [65], while adjusting several parameters. In the work of Liang et al. [65], *loading* and *inference* time are separated, focusing on inference time. However, for our approach such a separation is difficult as loading and inference time may overlap.

We evaluate our approach considering *default negation* by applying the *win-not-win* test and merge *large (anti-)join tests* with *datalog recursion* and *default negation*, creating a new test called *transitive closure with negation*. Other metrics of Liang et al. [65], such as *indexing*, are not supported by the MapReduce framework, while all optimizations and cost-based analysis were performed manually.

**Platform.** We have implemented our experiments using the Hadoop MapReduce framework<sup>1</sup>, version 1.2.1. We have performed experiments on a cluster of the University of Huddersfield. The cluster consists of 8 nodes (one node was allocated as “master” node), using a Gigabit Ethernet interconnect. Each node was equipped with 4 cores running at 2.5GHz, 8GB RAM and 250GB of storage space.

**Evaluation tests.** The *win-not-win* test, presented by Liang et al. [65], consists of a single rule, where *move* is the base relation:

$$\text{win}(X) \leftarrow \text{move}(X, Y), \text{not win}(Y).$$

We test the following data distributions:

<sup>1</sup><http://hadoop.apache.org/mapreduce/>

- the base facts form a cycle:  $\{\text{move}(1,2), \dots, \text{move}(i, i+1), \dots, \text{move}(n-1,n), \text{move}(n,1)\}$ .
- the data is tree-structured:  $\{\text{move}(i, 2^*i), \text{move}(i, 2^*i+1) \mid 1 \leq i \leq n\}$ .

We used four cyclic datasets and four tree-structured datasets with 125M, 250M, 500M and 1000M facts.

The *transitive closure with negation* test consists of the following rule set, where  $b$  is the base relation:

$$\begin{aligned} \text{tc}(X,Y) &\leftarrow \text{par}(X,Y). \\ \text{tc}(X,Y) &\leftarrow \text{par}(X,Z), \text{tc}(Z,Y). \\ \text{par}(X,Y) &\leftarrow b(X,Y), \mathbf{not} \ q(X,Y). \\ \text{par}(X,Y) &\leftarrow b(X,Y), b(Y,Z), \mathbf{not} \ q(Y,Z). \\ q(X,Y) &\leftarrow b(Z,X), b(X,Y), \mathbf{not} \ q(Z,X). \end{aligned}$$

We test the following data distribution:

- the base facts are chain-structured:  $\{b(i, i+k) \mid 1 \leq i \leq n, k < n\}$ . Intuitively, the  $i$  values are distributed over  $\lceil n/k \rceil$  levels, allowing  $\lceil n/k \rceil - 1$  joins in the formed chain.

The *transitive closure with negation* test allows for comparing the performance of the naive and the optimized WFS fixpoint calculation when the computation of  $\text{lfp}(T_{P,J}(I))$  starts from  $I = \emptyset$  and  $I \neq \emptyset$  respectively. For  $U_i$  and  $K_{i+1}$ , for  $i \geq 0$ , the optimized implementation speeds up the process by using, as input, the previously computed transitive closure of  $K_i$ , while the naive implementation comes with the overhead of recomputing previously inferred literals.

We used four chain-structured datasets for increasing number of joins in the initially formed chain ( $\lceil n/k \rceil - 1$ ) with  $n = 125\text{M}$ , and  $k = 41.7\text{M}, 25\text{M}, 13.9\text{M}$  and  $7.36\text{M}$ , and four chain-structured datasets for a constant number of joins in the initially formed chain ( $\lceil n/k \rceil - 1$ ) with  $n = 62.5\text{M}, 125\text{M}, 250\text{M}$  and  $500\text{M}$ , and  $k = 12.5\text{M}, 25\text{M}, 50\text{M}$  and  $100\text{M}$  respectively.

**Results.** We can identify four main factors that affect the performance of our approach:

1. **Number of facts**, affecting the input size.
2. **Number of rules**, affecting the output size.
3. **Data distribution**, affecting the number of required MapReduce jobs.
4. **Rule set structure**, affecting the number of required MapReduce jobs.

Figure 6.1 presents the runtimes of our system for the *win-not-win* test over cyclic datasets with input sizes up to 1 billion facts. In this case, our system scales linearly with respect to both dataset size and number of nodes. This is attributed to the fact that the



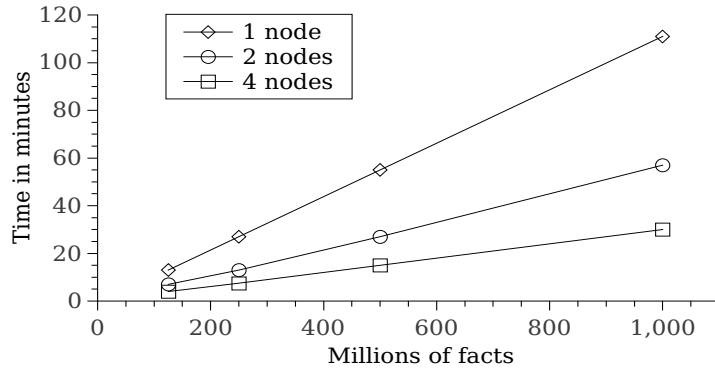


Figure 6.1: *Win-not-win* test for cyclic datasets. Time in minutes as a function of dataset size, for various numbers of nodes.

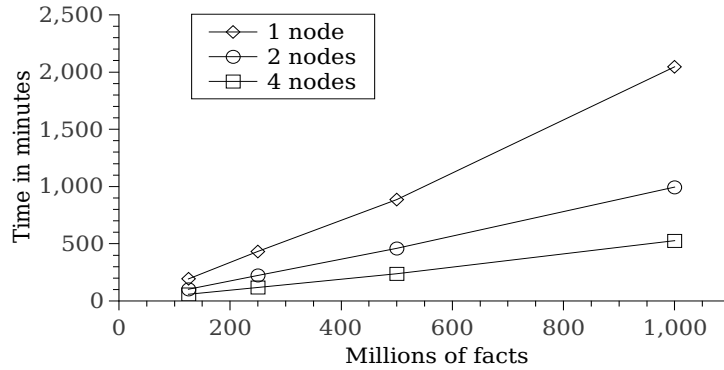


Figure 6.2: *Win-not-win* test for tree-structured datasets. Time in minutes as a function of dataset size, for various numbers of nodes.

runtime per MapReduce job scales linearly for increasing data sizes, while the number of jobs remains constant.

Figure 6.2 shows the runtimes of our system for the *win-not-win* test over tree-structured datasets with input sizes up to 1 billion facts. Our approach scales linearly for increasing data sizes and number of nodes. It is evident that while all other factors remain stable (rule set, number of facts and number of nodes), different data distributions may result in completely different runtimes (see Figure 6.1 and Figure 6.2).

Figure 6.3 depicts the scaling properties of our system for the *transitive closure with negation* test over chain-structured datasets, when run on 7 nodes. Practically, transitive closure depends on the number of joins in the initially formed chain, which are equal to  $\lceil n/k \rceil - 1$ , namely 2, 4, 8 and 16, and thus, appropriate for scalability evaluation. The length of the chain affects both the size of the transitive closure and the number of “inference steps”, leading to polynomial complexity. Note that our results are in line with Theorem 2.1.1. Finally, the speedup of the optimized over the naive implementation is

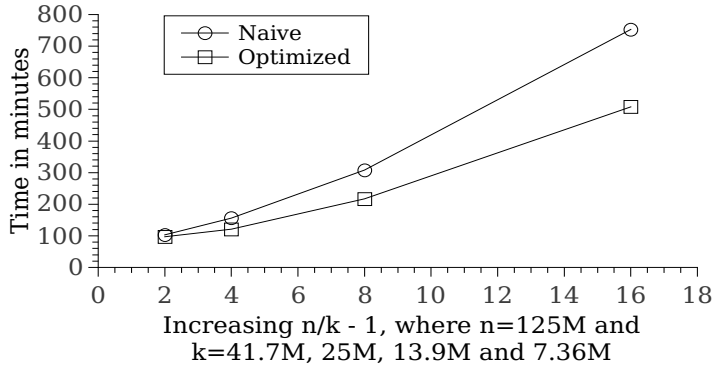


Figure 6.3: *Transitive closure with negation* test for chain-structured datasets. Time in minutes as a function of number of joins in the initially formed chain ( $\lceil n/k \rceil - 1$ ), for dataset size ( $n$ ) and number of facts per level ( $k$ ), comparing naive and optimized WFS fixpoint calculation.

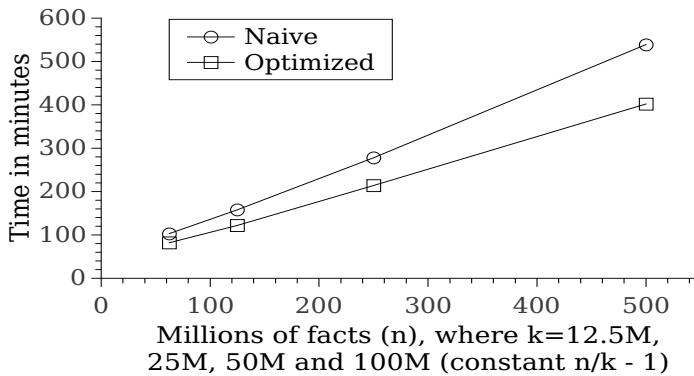


Figure 6.4: *Transitive closure with negation* test for chain-structured datasets. Time in minutes as a function of dataset size ( $n$ ) and number of facts per level ( $k$ ) (constant  $\lceil n/k \rceil - 1$ ), comparing naive and optimized WFS fixpoint calculation.

higher for longer chains, since the naive implementation has to recompute larger transitive closures.

Figure 6.4 illustrates the scalability properties of our system for the *transitive closure with negation* test over chain-structured datasets for constant number of joins in the initially formed chain, when run on 7 nodes. Our approach scales linearly, both for naive and optimized implementation as the number of jobs remains constant, while the runtime per job scales linearly for increasing number of facts.

## 6.2 Stratified Versus Full Well-Founded Semantics Approach

In order to justify our claim about the computational advantages of the presented method for the stratified well-founded semantics over the method for the full well-founded semantics, we provide here a theoretical and an experimental analysis comparing both methods over stratified programs.

### 6.2.1 Theoretical Analysis

It is evident that for a stratified program, predicate dependencies are a key attribute affecting the reasoning process. Indeed, for the stratified WFS method, the longest dependency chain affects the number of ranks, which in turn defines the rules application sequence. Thus, longer dependency chains result in higher numbers of ranks. On the other hand, for the general case of the full WFS method, there is no clear reasoning sequence since both rule set and dataset distribution affect the number of “inference steps”.

By examining closely several stratified programs over the full WFS method it becomes evident that, for stratified programs, the main factor affecting the number of “inference steps” is the longest dependency chain. In fact, the progress that is made during each “inference steps”, namely more knowledge of true (known) facts ( $K_m \subseteq K_{m+1}$ ,  $m \geq 0$ ) and a better estimation of possible (unknown) facts ( $U_m \supseteq U_{m+1}$ ,  $m \geq 0$ ), follows closely the progressive reasoning process of the stratified WFS method. Thus, the two estimations (the sets of literals  $K$  and  $U$ ) gradually “converge”, namely part of the set remains constant, on predicates belonging initially to lower ranks and consecutively to higher ranks.

For the stratified WFS method, the definition of ranks provides a well-defined reasoning sequence with each rule being applied only once. Thus, in order to further highlight the computational advantages of the stratified over the full WFS method, we will define a rule set that will allow the computation of the least fixpoint, namely the sets of literals  $K$  and  $U$ , by computing the rules only once. In this way, we eliminate the overhead of computing the least fixpoint by applying rules until no new knowledge is derived, which may require several iterations over the entire rule set.

In order to highlight the performance differences between the two methods, we define a rule set that forms a single dependency chain, where  $b$  is the base relation and  $i \geq 1$ :

$$a_i(X,Y) \leftarrow b(X,Y), \text{ not } a_{i-1}(X,Y).$$

over the following distribution:

- the base facts are uniform:  $\{b(j, j) \mid 1 \leq j \leq n\}$ . Intuitively, since the dataset contains facts only for  $b(X, Y)$ , the aforementioned rule set will produce:
  - **true** literals, for  $a_i(X, Y)$ , where  $i$  is an odd number.

- no **undefined** literals, since the program is stratified.
- **false** literals, for  $a_i(X, Y)$ , where  $i$  is an even number. Recall that **false** literals are not included in the knowledge base.

Consider the following program:

$$a_1(X, Y) \leftarrow b(X, Y), \text{ not } a_0(X, Y).$$

and the following fact:

$$b(1, 1)$$

The stratified WFS method requires a single application of the aforementioned rule, since the program predicates are assigned to ranks as follows:

$$\begin{aligned} \text{rank 0: } & a_0, b \\ \text{rank 1: } & a_1 \end{aligned}$$

In general, for  $i = N$ , each rule needs to be applied only once, applying rules from rank 1 to rank  $N$ , requiring a total of  $N$  rule computations, which will be executed over  $N$  MapReduce jobs. On average, the total input is  $N * (n + n/2)$ , where  $N$  is the number of ranks and  $n$  is the number of facts (note that we include  $n$  facts for  $b$  and an average of  $n/2$  facts for each  $a_i$ ). On average, the total output is  $N * (n/2)$ , where  $N$  is the number of ranks and  $n$  is the number of facts.

On the other hand, the full WFS method requires several “inference steps” in order to reach a fixpoint, with sets of literals  $K$  and  $U$  being as follows (for  $i = 1$ , namely the rule set consists of one rule):

$$\begin{aligned} K_0 &= \{b(1, 1)\} \\ U_0 &= \{a_1(1, 1), b(1, 1)\} \\ K_1 &= \{a_1(1, 1), b(1, 1)\} : +a_1(1, 1) \\ U_1 &= \{a_1(1, 1), b(1, 1)\} \\ K_2 &= \{a_1(1, 1), b(1, 1)\} = K_1 \end{aligned}$$

where “ $: +a_1(1, 1)$ ” denotes that  $K_1 - K_0 = \{a_1(1, 1)\}$ , while a fixpoint is reached at step 2 since  $K_1 = K_2$ .

For  $i = 2$  (namely the rule set consists of two rules), the sets of literals  $K$  and  $U$  are as follows:

$$\begin{aligned} K_0 &= \{b(1, 1)\} \\ U_0 &= \{a_2(1, 1), a_1(1, 1), b(1, 1)\} \\ K_1 &= \{a_1(1, 1), b(1, 1)\} : +a_1(1, 1) \\ U_1 &= \{a_1(1, 1), b(1, 1)\} : -a_2(1, 1) \\ K_2 &= \{a_1(1, 1), b(1, 1)\} = K_1 \end{aligned}$$

where “: +a<sub>1</sub>(1,1)” denotes that  $K_1 - K_0 = \{a_1(1,1)\}$ , “: -a<sub>2</sub>(1,1)” denotes that  $U_0 - U_1 = \{a_2(1,1)\}$ , while a fixpoint is reached at step 2 since  $K_1 = K_2$ .

In general, for  $i = N$  and  $n$  facts for  $b(X,Y)$ , the full WFS method computes an initial estimation during step 0, while the computation of the sets of literals  $K$  and  $U$  gradually refines the estimation until fixpoint is reached. For the aforementioned rule set and dataset:

$$\begin{aligned} \lceil (N+3)/2 \rceil \text{ steps are required, if } i \text{ is an even number} \\ \lceil (N+4)/2 \rceil \text{ steps are required, if } i \text{ is an odd number} \end{aligned}$$

Note that the least fixpoint computes each set of literals  $K$  and  $U$ , thus:

$$\begin{aligned} (N+3) \text{ least fixpoints are computed, if } i \text{ is an even number} \\ (N+4) \text{ least fixpoints are computed, if } i \text{ is an odd number} \end{aligned}$$

We need to apply all  $N$  rules in order to compute each least fixpoint. Note that no rules need to be applied in order to compute the least fixpoint for  $K_0$  since the given rule set contains only negative rules, thus:

$$\begin{aligned} (N+2) * N \text{ total rule applications are required, if } i \text{ is an even number} \\ (N+3) * N \text{ total rule applications are required, if } i \text{ is an odd number} \end{aligned}$$

with  $N$  rules being applied within a single MapReduce job:

$$\begin{aligned} (N+2) \text{ MapReduce jobs are required, if } i \text{ is an even number} \\ (N+3) \text{ MapReduce jobs are required, if } i \text{ is an odd number} \end{aligned}$$

with total input being (until fixpoint is reached):

$$\begin{aligned} (N+2) * (n + N * (n/2)), \text{ if } i \text{ is an even number} \\ (N+3) * (n + N * (n/2)), \text{ if } i \text{ is an odd number} \end{aligned}$$

and total output being (until fixpoint is reached):

$$\begin{aligned} (N+2) * N * (n/2), \text{ if } i \text{ is an even number} \\ (N+3) * N * (n/2), \text{ if } i \text{ is an odd number} \end{aligned}$$

Thus, the estimated cost of the full WFS method is higher compared to the stratified WFS method, since the full WFS method requires more MapReduce jobs in order to reach a fixpoint, while each MapReduce job has larger input and output, which in turn translates into longer execution time. Note that for more complex rule sets, the computation of each least fixpoint may require an iteration over the entire rule set, thus, further increasing the cost of the full WFS method compared to the stratified WFS method.

### 6.2.2 Experimental Analysis

**Methodology.** As described above, we need to propose stratified programs that would allow the evaluation of both the stratified and the full WFS method. In addition, the proposed stratified programs should minimize the overhead of the full WFS method, namely

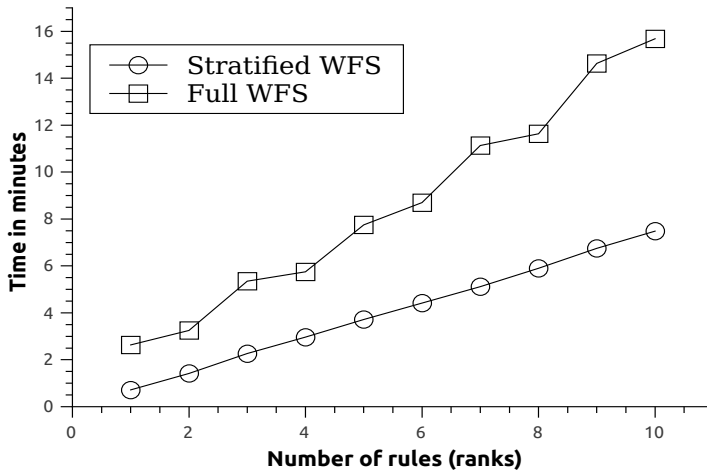


Figure 6.5: Time in minutes as a function of number of rules (ranks), comparing the stratified and the full WFS method.

Table 6.1: Speedup of the stratified over the full WFS method.

Number of rules	1	2	3	4	5	6	7	8	9	10
Speedup	3.67	2.29	2.36	1.94	2.09	1.97	2.18	1.97	2.17	2.1

each least fixpoint should be computed within a single MapReduce job, instead of a sequence of jobs that apply rules until no new knowledge is derived. Moreover, a single dependency chain is defined in order to allow a transparent way to evaluate the reasoning process. Note that our evaluation favors the full WFS method so as to show that the stratified WFS method retains its performance advantages for any given stratified program.

**Platform.** We have implemented our experiments using the Hadoop MapReduce framework<sup>2</sup>, version 1.0.4. We have performed experiments on a PC equipped with 4 cores running at 2.4GHz, 6GB RAM and 500GB of storage space.

**Evaluation test.** We define the following rule set that forms a single dependency chain, where  $b$  is the base relation and  $i \geq 1$ :

$$a_i(X,Y) \leftarrow b(X,Y), \text{ not } a_{i-1}(X,Y).$$

over the following distribution:

- the base facts are uniform:  $\{b(j, j) \mid 1 \leq j \leq n\}$ .

**Results.** Figure 6.5 presents the runtimes for both the stratified and the full WFS method, over 1 million facts, ranging the number of rules from 1 to 10. It is evident that the stratified WFS method clearly outperforms the full WFS method for the evaluated

<sup>2</sup><http://hadoop.apache.org/mapreduce/>

programs. In terms of speedup, as shown in Table 6.1 the stratified WFS method provides a speedup between 1.94 and 3.67, which depends on the number of rules. It is evident that for the majority of the tested programs the stratified WFS method computes the closure 2 times faster. Note that, for rule sets that require a sequence of rule applications in order to compute each least fixpoint, the speedup is expected to be even higher.





## Chapter 7

# Conclusion and Future Work

In this chapter we provide our observations from a preliminary investigation on a restricted form of answer set programming, namely single variable answer set programs. Subsequently, a generic evaluation framework for large-scale reasoning is presented, followed by a discussion of the main findings of this work and opportunities for future work.

### 7.1 Answer Set Programming

#### 7.1.1 Single Variable Programs

In this section, we present a solution for single variable ASP programs. Consider the following program:

$$\begin{aligned} \text{male}(X) &\leftarrow \text{human}(X), \text{ not } \text{female}(X). \\ \text{female}(X) &\leftarrow \text{human}(X), \text{ not } \text{male}(X). \end{aligned}$$

and the following facts:

$$\text{human}(\textit{pat}) \quad \text{human}(\textit{mary}) \quad \text{female}(\textit{mary})$$

The name *Pat* may either refer to a male (Patrick) or a female (Patricia), while *Mary* is a well known female name. Thus, this program has two answer sets, namely:

$$\begin{aligned} &\{\text{human}(\textit{pat}), \text{human}(\textit{mary}), \text{male}(\textit{pat}), \text{female}(\textit{mary})\} \\ &\{\text{human}(\textit{pat}), \text{human}(\textit{mary}), \text{female}(\textit{pat}), \text{female}(\textit{mary})\} \end{aligned}$$

A typical ASP solver would first ground the given program and then compute the aforementioned answer sets. However, this process can be improved for programs with one variable. The key observation here is the fact that for each rule, literals in both head and body are instantiated using the same constant (as  $X$  is the only variable). Since this is true for every rule, we can consider the following program for each constant separately:

---

**Algorithm 11** Parallel inference based on constants
 

---

```

map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: document line (a fact)
  constant = getConstant(value);
  predicate = getPredicate(value);
  EmitIntermediate(constant, predicate);

reduce(String key, Iterator values) :
  // program: propositional program stored in memory
  // key: constant
  // values : list of predicates (facts)
  Set facts =  $\emptyset$ ;
  for each  $v$  in values
    facts.add(v);
  ASPSolver solver = ASPSolver.getSolver();
  solver.reason(facts, program);
  Emit(key , solver.getResults());

```

---


$$\begin{aligned} \textit{male} &\leftarrow \textit{human}, \text{ not } \textit{female}. \\ \textit{female} &\leftarrow \textit{human}, \text{ not } \textit{male}. \end{aligned}$$

Thus, reasoning is performed for *pat* and *mary* separately. There are two answer sets for *pat*, namely  $\{\textit{human}(\textit{pat}), \textit{male}(\textit{pat})\}$  and  $\{\textit{human}(\textit{pat}), \textit{female}(\textit{pat})\}$ , and one answer set for *mary*, namely  $\{\textit{human}(\textit{mary}), \textit{female}(\textit{mary})\}$ .

Such reasoning process reduces both reasoning complexity and space requirements for answer sets storage. In general, we consider the following parameters:

- $k$ , which is the number of predicates.
- $m$ , which is the number of constants.
- $n = k * m$ , which is the number of ground literals.

For the program containing the variable  $X$ , there are  $2^n$  potential answer sets, since we need to consider every ground literal. However, by performing reasoning for each constant separately, the number of potential answer sets is reduced to  $2^k * m$ . In addition to this complexity reduction, we can speed up the process by performing reasoning, either on each constant or on each subset of predicates, in parallel applying the MapReduce framework.

### 7.1.2 Parallel Reasoning Based on Constants

In this section, we provide a solution based on the MapReduce framework. Our solution is based on the concept of grouping given facts based on their constants and performing rea-

soning for each constant separately, as described in Algorithm 11. Consider the following program:

$$\begin{aligned} male &\leftarrow human, \text{ not } female. \\ female &\leftarrow human, \text{ not } male. \end{aligned}$$

The *Map* function reads facts of the form  $predicate(constant)$  and emits pairs of the form  $\langle constant, predicate \rangle$ . Functions  $getConstant(value)$  and  $getPredicate(value)$  extract the constant and the predicate respectively from each *value* (given fact).

Given the facts:  $human(pat)$ ,  $human(mary)$  and  $female(mary)$ , *Map* function will emit the following pairs :

$$\begin{aligned} &\langle pat, human \rangle \\ &\langle mary, human \rangle \\ &\langle mary, female \rangle \end{aligned}$$

Note that, reasoning is performed for each constant (pat, mary) separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

$$\begin{aligned} &\langle pat, human \rangle \\ &\langle mary, \langle human, female \rangle \rangle \end{aligned}$$

Each *Reduce* function has to perform, in parallel, reasoning on an ASP solver (we assume any available solver that can be imported as an external library). First, *Reduce* function reads all the *values* (predicates) and adds them to *facts*, which is a set of predicates. We use a generic API defining functions such as  $ASPSolver.getSolver()$ , which provides an instance of the available solver. Subsequently,  $solver.reason(fact, program)$  performs reasoning based on the given predicates and the in-memory propositional program. Calculated answer sets are extracted from the solver using  $solver.getResults()$ .

Thus, during the reduce phase the reducer with key:

$$\begin{aligned} pat, & \text{ will emit } \langle pat, \langle \{human, male\}, \{human, female\} \rangle \rangle \\ mary, & \text{ will emit } \langle mary, \{human, female\} \rangle \end{aligned}$$

Note that in case there is no answer set for at least one constant, then no answer set exists for the given ASP program.

### 7.1.3 Parallel Reasoning Based on Predicates

An alternative solution for parallel reasoning, using the MapReduce framework, is based on the idea of grouping given facts according to the set of predicates they form for each constant and performing reasoning on each set of predicates separately. Consider the following program:

$$\begin{aligned} male &\leftarrow human, \text{ not } female. \\ female &\leftarrow human, \text{ not } male. \end{aligned}$$

**Algorithm 12** Calculating sets of predicates per constant

---

```

map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: document line (a fact)
  constant = getConstant(value);
  predicate = getPredicate(value);
  EmitIntermediate(constant, predicate);

reduce(String key, Iterator values) :
  // key: constant
  // values : list of predicates (facts)
  Set facts =  $\emptyset$ ;
  for each  $v$  in values
    facts.add(v);
  Emit(facts, key);

```

---

Our first step is to calculate the sets of predicates for each constant, as depicted in Algorithm 12. Given the facts: *human(pat)*, *human(mary)* and *female(mary)*, *Map* function will emit the following pairs :

$$\begin{aligned} &\langle \text{pat}, \text{human} \rangle \\ &\langle \text{mary}, \text{human} \rangle \\ &\langle \text{mary}, \text{female} \rangle \end{aligned}$$

Note that, the calculation of corresponding set of predicates for each constant (*pat*, *mary*) is performed separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

$$\begin{aligned} &\langle \text{pat}, \text{human} \rangle \\ &\langle \text{mary}, \langle \text{human}, \text{female} \rangle \rangle \end{aligned}$$

Each *Reduce* function has to calculate and emit, in parallel, the corresponding sets of predicates. Thus, *Reduce* function reads all the *values* (predicates) and adds them to *facts*, which is a set of predicates. Finally, both the set of predicates (*facts*) and the constant (*key*) are emitted.

During the reduce phase the reducer with key:

$$\begin{aligned} &\textit{pat}, \text{ will emit } \langle \text{human}, \text{pat} \rangle \\ &\textit{mary}, \text{ will emit } \langle \{\text{human}, \text{female}\}, \text{mary} \rangle \end{aligned}$$

Our next step is to perform reasoning for each set of predicates, as described in Algorithm 13. The *Map* function reads input of the form  $\langle \text{set of predicates}, \text{constant} \rangle$  and emits pairs of the form  $\langle \text{predicates}, \text{constant} \rangle$ . Functions *getPredicates(value)* and *getConstant(value)* extract the set of predicates and the constant respectively from each *value* (group of predicates per constant).

---

**Algorithm 13** Parallel inference based on predicates

---

```

map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: document line (a fact)
  predicates = getPredicates(value);
  constant = getConstant(value);
  EmitIntermediate(predicates, constant);

reduce(String key, Iterator values) :
  // program: propositional program stored in memory
  // key: predicates
  // values : list of constants
  ASPSolver solver = ASPSolver.getSolver();
  solver.reason(key, program);
  for each v in values
    Emit(v, solver.getResults());

```

---

Given as input the pairs:  $\langle \text{human}, \text{pat} \rangle$  and  $\langle \{\text{human}, \text{female}\}, \text{mary} \rangle$ , *Map* function will emit the following pairs :

$$\begin{aligned} &\langle \text{human}, \text{pat} \rangle \\ &\langle \{\text{human}, \text{female}\}, \text{mary} \rangle \end{aligned}$$

Note that, reasoning is performed for each set of predicates separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

$$\begin{aligned} &\langle \text{human}, \text{pat} \rangle \\ &\langle \{\text{human}, \text{female}\}, \text{mary} \rangle \end{aligned}$$

Each *Reduce* function has to perform, in parallel, reasoning on an ASP solver (we assume any available solver that can be imported as an external library). We use a generic API defining functions such as *ASPSolver.getSolver()*, which provides an instance of the available solver. Subsequently, *solver.reason(fact,program)* performs reasoning based on the given set of predicates and the in-memory propositional program. Finally, *Reduce* function reads all the *values* (constants) and emits, for each constant, the calculated answer sets. Calculated answer sets are extracted from the solver using *solver.getResults()*.

Thus, during the reduce phase the reducer with key:

$$\begin{aligned} &\text{human}, \text{ will emit } \langle \text{pat}, \langle \{\text{human}, \text{male}\}, \{\text{human}, \text{female}\} \rangle \rangle \\ &\{\text{human}, \text{female}\}, \text{ will emit } \langle \text{mary}, \{\text{human}, \text{female}\} \rangle \end{aligned}$$

Note that in case there is no answer set for at least one set of predicates, then no answer set exists for the given ASP program.

### 7.1.4 Final Remarks

Let us elaborate on the advantages of each parallelization technique and the challenges coming from programs containing rules with multiple variables. Practically, the parallelization technique that is more suitable in each case depends on the given program. Recall that the number of potential answer sets is reduced to  $2^k * m$ , where  $k$  is the number of predicates and  $m$  is the number of constants.

Parallel reasoning based on predicates has the advantage that reasoning for each set of predicates is performed only once. However, for programs containing few predicates, parallelization may be affected as reasoning process can be speeded up by a factor of  $2^k$ . On the other hand, for reasoning based on constants, given that the amount of constants is most likely to be larger than the amount of available nodes in the cluster, parallelization comes with higher scalability. Clearly, the best approach depends on the number of predicates and constants, and their distribution within the given program.

In future work, we intend to implement our approach and test it empirically. Currently this is somewhat hampered by a requirement of the Hadoop MapReduce framework, namely the ASP solver to be called should be written in Java, and its source should be available. So far, we were unable to find an existing solver implemented in Java with available source code. We will therefore have to defer experimental evaluation, as implementing a serial ASP solver in Java is beyond the scope of this thesis.

For programs containing rules with more than one variable, parallelization is more complex. It is evident that the proposed approach above cannot be directly generalized for programs that contain more than one variable since such programs, in general, cannot be partitioned in independent segments. Facts are most likely to be included in several overlapping subsets, while potential answer sets may be overlapping as well, resulting in the re-computation of certain parts of the reasoning process. Further investigation is required in order to devise a scalable and efficient mechanism for parallel ASP reasoning.

To sum up the above, we provide a list of key theoretical and practical results:

- For single variable ASP programs, complexity can be reduced from  $2^n$  to  $2^k * m$ , where  $k$  is the number of predicates,  $m$  is the number of constants, and  $n$  is the number of ground literals.
- Single variable ASP programs can be parallelized either on constants or on predicates, where the most efficient approach depends on the given program.
- An implementation based on MapReduce requires an external ASP solver implemented in Java. Given that currently there is no ASP solver implemented in Java, one may consider applying the proposed method using a different computing model, such as OpenMP or MPI.
- The proposed method cannot be directly extended to ASP programs with multiple variables.

## 7.2 Evaluation Framework

Here we present our conclusions with respect to experimental evaluation of large-scale parallel reasoning. Based on our experience, a key attribute is the need to reassure that the proposed implementation is scalable. Thus, several aspects of the implementation should be evaluated according to the following metrics.

**Dataset.** The proposed approach should be evaluated over suitable datasets. Two main aspects that should be evaluated are: (a) scalability over increasing data sizes and (b) workload distribution over several data distributions. Datasets may follow diverse distributions ranging from perfectly even distribution to highly skewed data distribution. As described in the work of Kotoulas et al. [33] and Duan et al. [62], the fact that Semantic Web data are highly skewed may result, if not addressed properly, in low degree of parallelization. In case real-world datasets are not available, one may use either benchmarks such as LUBM<sup>1</sup> or manually generated datasets resembling the required distribution.

**Rule set.** Each approach should be evaluated over several rule sets. Two main aspects that should be evaluated are: (a) scalability over increasing number of rules and (b) system performance over a variety of rule set structures. For the case of Semantic Web reasoning, RDFS and OWL-Horst rule sets have been used in the literature. However, for more general approaches such as defeasible reasoning, manually generated rule sets are required due to the absence of available benchmarks. In each case, we need to ensure that the mixture of the evaluated rule sets covers a wide range of domains of application.

**Platform.** In order to evaluate parallel reasoning, we need to consider a platform that is able to run either on a cluster configuration or on the cloud since the system should be evaluated over increasing number of nodes. In addition, the selected platform should be able to handle huge amounts of data, while offering scalability for up to hundreds or thousands nodes. In this work we mainly used the Hadoop MapReduce framework<sup>2</sup> which can serve as a highly distributed platform since it is designed to process huge amounts of data by coordinating up to several thousands nodes.

**Evaluation Settings.** In a nutshell a system can be evaluated over the following parameters:

- **Runtime**, as the time required to calculate the inferential closure of the input.
- **Number of nodes** performing the computation in parallel.
- **Dataset size**, expressed in the number of facts in the input.
- **Dataset distribution**, describing whether dataset follows even or uneven distribution.
- **Rule set size**, expressed in the number of rules for a given rule set.

---

<sup>1</sup><http://swat.cse.lehigh.edu/projects/lubm/>

<sup>2</sup><http://hadoop.apache.org/mapreduce/>

- **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.
- **Compression ratio**, affecting output size, when compression is used.

### 7.3 Discussion

This work is the first to explore the feasibility of large-scale nonmonotonic reasoning over huge data sets. In contrast to previously existing works, we dealt with reasoning over inconsistent or missing information, which requires computationally more complex models. In particular, we studied the following nonmonotonic logics: defeasible logic (see Chapter 4), well-founded semantics (see Chapters 5 and 6) and answer set programming (see Section 7.1). Each logic provides a more expressive framework, compared to monotonic reasoning, for processing available datasets.

**Defeasible logic** provides an approach for dealing with inconsistency as it supports rule prioritization, where newly derived knowledge comes from conflict resolution by applying the defeasible logic algorithm. In particular, for conflicting conclusions, defeasible logic examines which rules are applicable (see Section 4.2.2) and determines the final derivations based on the provided rule priorities (see Section 4.2.3). In a sense, conclusions are separated into conflicting teams, with the team winning the dispute also leading to new conclusions.

We showed that full defeasible logic can be applied for rule sets that contain one variable (see Section 4.1). In this setting, facts are grouped based on their corresponding value for the variable with each group of predicates being fed to a defeasible reasoner for conclusion derivation. This restricted form of rule sets is fully parallelizable using the MapReduce framework and can handle billions of facts, providing linear scalability with respect to both dataset size and number of nodes (for more details see Tachmazidis et al. [12]). However, there was a clear need for a more general approach since most applications would require rule sets containing more than one variable.

Our next step was to devise an approach for rule sets with predicates of arbitrary arity (see Section 4.2). Here we had to consider two types of rule sets, namely stratified and non-stratified. We presented a solution for stratified rule sets, where rules are grouped based on conclusion dependencies. In contrast to monotonic reasoning, fired rules are not directly materialized. Defeasible reasoning is based on recording fired rules (see Section 4.2.2) and combining them with existing facts, thus dealing with arising inconsistencies (see Section 4.2.3). The presented approach is scalable for datasets of up to 1 billion facts coming with good scalability properties (see Section 4.3).



Defeasible reasoning can extend existing monotonic approaches by providing an additional layer of reasoning. In this way, monotonic reasoning is initially performed, while defeasible reasoning subsequently enriches the knowledge base and resolves arising conflicts. However, a generic solution, having the ability to support defeasible reasoning over non-stratified rule sets, is yet to be defined (see Section 4.2.4). An existing serial defeasible algorithm could be directly parallelized. Nonetheless, it generates an excessive amount of information, which is prohibitive in the face of Big Data. It remains an open question whether an approach can be devised, supporting full defeasible logic derivation and retaining scalability by generating a manageable amount of information.

**Well-Founded Semantics** is a prominent form of reasoning which allows for conclusion derivation over missing information (see Chapters 5 and 6). Specifically, the body of each rule may contain both positive and negative subgoals (see Sections 5.1 and 6.1.1). Thus, a conclusion is derived when part of the information is found in the knowledge base while some information is not included or missing. However, the crucial difference is that recursion through negation is allowed, meaning that the well-founded model can contain undefined atoms. Thus, each literal is classified as true, undefined or false (see Chapter 6).

This type of classification, namely literals are assigned one of the three values, pose a significant scalability barrier as storing all three values would result in storing the entire Herbrand base, which is prohibitive for Big Data. Thus, we started our investigation from stratified rule sets, initially overcoming the challenge of computing, in parallel, rules that contain both positive and negative subgoals (see Chapter 5). Note that existing approaches allowed rules containing only positive subgoals. By restricting the set of allowed rules to safe rules, namely rules where each variable in the head of the rule also occurs in a positive subgoal, we managed to propose a parallel and scalable approach for stratified rule sets, where reasoning follows a linear scalability for increasing number of nodes and number of facts (see Section 5.2).

The main challenge was still present as we could not compute the full well-founded semantics for a given dataset. The solution came by applying the alternating fixpoint procedure (see Sections 2.1.9 and 6.1) which allowed the full materialization of the well-founded semantics by reasoning over and storing only true and unknown literals. As we showed this approach generates a manageable amount of data and comes with good scalability properties for datasets of up to 1 billion facts (see Section 6.1.3). In addition, we utilized several theoretical properties that were introduced in the literature in order to provide an optimized implementation (see Section 6.1.2). It is evident that the optimized approach provides better results as it minimizes the recalculation of already computed information (see Section 6.1.3).

**Answer Set Programming** (ASP) was probably the most challenging approach to tackle. Thus, we initially focused on a restricted form called monadic programs, namely programs where all predicates are of arity one (see Section 7.1.1). Two solutions were proposed. The first solution is based on parallelizing on constants (see Section 7.1.2),

where facts are grouped based on the value of the argument for the given predicate. Thus, each constant contains all predicates that are fed to a standard ASP solver, which is called as an external library. The main advantage of this solution is the fact that the number of constants is expected to be larger than the number of nodes, thus providing high degree of parallelization.

The second solution is based on parallelizing on predicates (see Section 7.1.3). Once facts are grouped on constants it may become evident that several constants contain the exact same set of predicates. This leads to a repetitive computation of the exact same answer sets for several constants. Thus, we regroup facts based on predicates. In this way, the answer sets for each set of predicates are computed once and emitted for all corresponding constants. We provide a theoretical analysis for each proposed solution, while experimental evaluation was deferred due to the lack of existing ASP solver written in Java.

By dealing with monadic programs we managed to comprehend the complexity of the general case where alternative world views are computed, namely a given program may have none, one or more than one answer sets. In addition, answer sets may also be overlapping, implying that certain parts may need to be recomputed. The presented approach cannot be directly extended to more general programs as it is based on the hypothesis that the given program can be partitioned in independent segments. Thus, a different direction is required in order to provide a parallel and scalable solution for the general case.

**Final remarks.** The aforementioned approaches have been developed and tested mainly over the MapReduce framework. However, a closer examination reveals the fact that the proposed methods can be implemented using other parallel and distributed methods such as OpenMP<sup>3</sup>, Message Passing Interface (MPI)<sup>4</sup>, Graphics Processing Units (GPUs) [66], shared memory and distributed memory supercomputers. In fact, our approach for the well-founded semantics has been implemented and evaluated on top of X10 using a distributed memory setting [17].

Note that the lack of a scalable method for the full defeasible logic and a more general case of the answer set programming is not attributed to a given parallel programming model, such as the MapReduce framework. In order to overcome existing limitations, a better understanding of the underlying theory is required and a more elaborate method, from a theoretical perspective, should be devised. For the case of defeasible logic, a more compact reasoning process is required that will eventually allow closure computation, without the need for Herbrand base instantiation. For the case of answer set programming, the main issue is the efficient handling of alternative worlds, namely evaluating alternative paths towards a solution, in parallel, may not necessarily lead to better performance due to the excessive searching space size, with currently known efficient approaches being mainly

---

<sup>3</sup><http://openmp.org/wp/>

<sup>4</sup><http://www.mcs.anl.gov/research/projects/mpi/>

based on heuristics.

## 7.4 Future Work

Our study on large-scale reasoning through mass parallelization can be extended in various dimensions, which are discussed below.

- As mentioned above, for the case of defeasible reasoning, an approach that would support the full defeasible logic is yet to be defined. Such a solution should be based on two key concepts, namely rule application and conclusion derivation should be parallelizable, while the application of defeasible algorithm should require the generation of manageable amount of data.
- An e-commerce application may be developed on top of defeasible logic, providing a matching mechanism for buyer's preferences and seller's offers. The system may resolve arising conflicts and provide useful suggestions in order to facilitate the trading process.
- Reasoning over the well-founded semantics have been studied based on forward chaining, thus computing the full materialization. However, a scalable query-based approach for the well-founded semantics is another prominent research direction.
- The presented solution for the well-founded semantics may serve as a basis for the implementation of a scalable approach for decision making in a smart environment setting [6]. In particular, the implemented system (see Nieves et al. [6]) may benefit from an efficient construction of arguments that will have the ability to operate over big data.
- Scalable computation of answer sets remains an open question mainly due to the complexity of the reasoning process. Thus, a different approach (compared to the one presented in this work) should be followed in order to realize this type of reasoning.
- The absence of existing and well-established benchmarks is clear from this work. In experimental evaluation, we needed to either generate synthetic data or adapt existing benchmarks. However, none of the existing benchmarks is designed to provide a sufficient evaluation of a nonmonotonic approach on the large-scale. Thus, a novel benchmark should be proposed with the ability to evaluate a given approach by testing various rule set sizes and levels of complexity, and various dataset sizes and distributions. Ideally, a thorough examination of existing real-world datasets would reveal the specifications for such novel benchmark.
- Tools and systems having the ability to handle inconsistency could be built based on the proposed solutions for either defeasible logic or the well-founded semantics.

As we showed for defeasible logic, the proposed approach can be used on top of already existing reasoners, extending the derivation process and further enriching the knowledge base.

- This work can be integrated in several use cases: (a) *smart cities* which generate huge amounts of data, (b) *social media* which may provide data that lead to conflicts, and (c) *eHealth* by dealing with medical records which are considered as sensitive data and therefore privacy should be reassured.

# Bibliography

- [1] A. Bikakis and G. Antoniou, “Contextual Defeasible Logic and Its Application to Ambient Intelligence,” *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 41, no. 4, pp. 705–716, 2011.
- [2] J. Du, G. Qi, J. Z. Pan, and Y. Shen, “A decomposition-based approach to OWL DL ontology diagnosis,” in *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*. IEEE Computer Society, 2011, pp. 659–664. [Online]. Available: <http://dx.doi.org/10.1109/ICTAI.2011.104>
- [3] G. Flouris, G. Konstantinidis, G. Antoniou, and V. Christophides, “Formal foundations for RDF/S KB evolution,” *Knowl. Inf. Syst.*, vol. 35, no. 1, pp. 153–191, 2013.
- [4] Y. Roussakis, G. Flouris, and V. Christophides, “Declarative Repairing Policies for Curated KBs,” in *HDMS*, 2011.
- [5] G. Antoniou and F. van Harmelen, *A semantic web primer*. MIT Press, 2004.
- [6] J. C. Nieves and H. Lindgren, “Deliberative argumentation for service provision in smart environments,” in *Multi-Agent Systems*. Springer, 2014, pp. 388–397.
- [7] G. Antoniou and M.-A. Williams, *Nonmonotonic reasoning*. MIT Press, 1997.
- [8] M. Knorr, P. Hitzler, and F. Maier, “Reconciling OWL and Non-monotonic Rules for the Semantic Web,” in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., vol. 242. IOS Press, 2012, pp. 474–479.
- [9] T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer, “Well-founded semantics for description logic programs in the semantic web,” *ACM Trans. Comput. Log.*, vol. 12, no. 2, p. 11, 2011.
- [10] G. Antoniou and A. Bikakis, “DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web,” *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 2, pp. 233–245, 2007.

- [11] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller, “Efficient Defeasible Reasoning Systems,” *IJAIT*, vol. 10, p. 2001, 2001.
- [12] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, “Towards parallel non-monotonic reasoning with billions of facts,” in *KR*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, 2012.
- [13] I. Tachmazidis, G. Antoniou, G. Flouris, S. Kotoulas, and L. McCluskey, “Large-scale Parallel Stratified Defeasible Reasoning,” in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., vol. 242. IOS Press, 2012, pp. 738–743.
- [14] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, “Scalable Nonmonotonic Reasoning over RDF Data Using MapReduce,” in *SSWS+HPCSW*, 2012.
- [15] I. Tachmazidis and G. Antoniou, “Computing the Stratified Semantics of Logic Programs over Big Data through Mass Parallelization,” in *RuleML*, ser. Lecture Notes in Computer Science, L. Morgenstern, P. S. Stefaneas, F. Lévy, A. Wyner, and A. Paschke, Eds., vol. 8035. Springer, 2013, pp. 188–202.
- [16] I. Tachmazidis, G. Antoniou, and W. Faber, “Efficient computation of the well-founded semantics over big data,” *TPLP*, vol. 14, no. 4-5, pp. 445–459, 2014. [Online]. Available: <http://dx.doi.org/10.1017/S1471068414000131>
- [17] I. Tachmazidis, L. Cheng, S. Kotoulas, G. Antoniou, and T. E. Ward, “Massively parallel reasoning under the well-founded semantics using X10,” in *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, 2014, pp. 162–169. [Online]. Available: <http://dx.doi.org/10.1109/ICTAI.2014.33>
- [18] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” in *OOPSLA*, 2005, pp. 519–538.
- [19] I. Tachmazidis, G. Antoniou, and W. Faber, “Computing Answer Sets for Monadic Logic Programs via Mapreduce,” in *ASPOCP*, 2014.
- [20] G. Antoniou, J. Z. Pan, and I. Tachmazidis, “Large-scale complex reasoning with semantics: Approaches and challenges,” in *Web Information Systems Engineering - WISE 2013 Workshops - WISE 2013 International Workshops BigWebData, MBC, PCS, STeH, QUAT, SCEH, and STSC 2013, Nanjing, China, October 13-15, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, Z. Huang, C. Liu, J. He, and G. Huang, Eds., vol. 8182. Springer, 2013, pp. 1–10. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-54370-8\\_1](http://dx.doi.org/10.1007/978-3-642-54370-8_1)

- [21] F. Baader, I. Horrocks, and U. Sattler, “Description Logics,” in *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 2008, ch. 3, pp. 135–180. [Online]. Available: [download/2007/BaHS07a.pdf](#)
- [22] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
- [23] 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.
- [24] A. V. Gelder, K. A. Ross, and J. S. Schlipf, “The well-founded semantics for general logic programs,” *J. ACM*, vol. 38, no. 3, pp. 620–650, 1991.
- [25] J.-M. Nicolas, “Logic for improving integrity checking in relational data bases,” *Acta Informatica*, vol. 18, pp. 227–253, 1982.
- [26] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995. [Online]. Available: <http://www-cse.ucsd.edu/users/vianu/book.html>
- [27] S. Brass, J. Dix, B. Freitag, and U. Zukowski, “Transformation-based bottom-up computation of the well-founded model,” *Theory and Practice of Logic Programming*, vol. 1, no. 5, pp. 497–538, 2001.
- [28] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
- [29] V. Lifschitz, “Answer set programming and plan generation,” *Artif. Intell.*, vol. 138, no. 1-2, pp. 39–54, 2002. [Online]. Available: [http://dx.doi.org/10.1016/S0004-3702\(02\)00186-8](http://dx.doi.org/10.1016/S0004-3702(02)00186-8)
- [30] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in *Proceedings of the 6th conference on Symposium on Operating 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>
- [31] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen, “Marvin: Distributed reasoning over large-scale Semantic Web data,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 4, pp. 305–316, 2009.
- [32] —, “MARVIN: A platform for large-scale analysis of Semantic Web data,” 2009.
- [33] S. Kotoulas, E. Oren, and F. van Harmelen, “Mind the data skew: distributed inferencing by speeddating in elastic regions,” in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April*

- 26-30, 2010, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 531–540. [Online]. Available: <http://doi.acm.org/10.1145/1772690.1772745>
- [34] E. L. Goodman, E. Jimenez, D. Mizell, S. al Saffar, B. Adolf, and D. Haglin, “High-performance Computing Applied to Semantic Databases,” in *Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications - Volume Part II*, ser. ESWC’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 31–45.
- [35] E. L. Goodman and D. Mizell, “Scalable In-memory RDFS Closure on Billions of Triples,” in *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, 2010, p. 17.
- [36] J. Weaver and J. A. Hendler, “Parallel materialization of the finite rdfs closure for hundreds of millions of triples,” in *International Semantic Web Conference*, 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. 682–697.
- [37] N. Heino and J. Z. Pan, “Rdfs reasoning on massively parallel hardware,” in *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I*, ser. ISWC’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 133–148.
- [38] M. Salvadores, G. Correndo, S. Harris, N. Gibbins, and N. Shadbolt, “The design and implementation of minimal rdfs backward reasoning in 4store,” in *ESWC (2)*, ser. Lecture Notes in Computer Science, G. Antoniou, M. Grobelnik, E. P. B. Simperl, B. Parsia, D. Plexousakis, P. D. Leenheer, and J. Z. Pan, Eds., vol. 6644. Springer, 2011, pp. 139–153.
- [39] —, “4sr - Scalable Decentralized RDFS Backward Chained Reasoning,” in *ISWC Posters&Demos*, 2010.
- [40] M. Salvadores, G. Correndo, T. Omitola, N. Gibbins, S. Harris, and N. Shadbolt, “4s-reasoner: Rdfs backward chained reasoning support in 4store,” in *Web-scale Knowledge Representation, Retrieval, and Reasoning (Web-KR3)*, September 2010, event Dates: September 2010. [Online]. Available: <http://eprints.soton.ac.uk/271255/>
- [41] S. Harris, N. Lamb, and N. Shadbolt, “4store: The design and implementation of a clustered rdf store,” in *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.
- [42] J. Hoeksema and S. Kotoulas, “High-performance Distributed Stream Reasoning using S4,” in *Proceedings of the 1st International Workshop on Ordering and Reasoning*, 2011.



- [43] R. Soma and V. K. Prasanna, "Parallel Inferencing for OWL Knowledge Bases," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 75–82.
- [44] 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.
- [45] J. Urbani, S. Kotoulas, J. Maassen, N. Drost, F. Seinstra, F. Van Harmelen, and H. Bal, "WebPIE: a Web-scale Parallel Inference Engine."
- [46] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal, "WebPIE: A Web-scale Parallel Inference Engine using MapReduce," *J. Web Sem.*, vol. 10, pp. 59–75, 2012.
- [47] —, "OWL reasoning with WebPIE: Calculating the Closure of 100 Billion Triples," in *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*, 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. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-13486-9\\_15](http://dx.doi.org/10.1007/978-3-642-13486-9_15)
- [48] A. Hogan, J. Z. Pan, A. Polleres, and S. Decker, "SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples," in *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science, P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, Eds., vol. 6496. Springer, 2010, pp. 337–353. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-17746-0\\_22](http://dx.doi.org/10.1007/978-3-642-17746-0_22)
- [49] K. Wu and V. Haarslev, "A Parallel Reasoner for the Description Logic ALC," in *Proceedings of the 2012 International Workshop on Description Logics, DL-2012, Rome, Italy, June 7-10, 2012*, ser. CEUR Workshop Proceedings, vol. 846. CEUR-WS.org, 2012.
- [50] T. Liebig and F. Müller, "Parallelizing tableaux-based description logic reasoning," in *On the Move to Meaningful Internet Systems 2007: OTM 2007*, ser. Lecture Notes in Computer Science, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 4806. Springer, 2007, pp. 1135–1144.
- [51] T. Liebig, A. Steigmiller, and O. Noppens, "Scalability via Parallelization of OWL Reasoning," in *Proceedings of the 4th International Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic (NeFoRS 2010)*, 2010.

- [52] A. Schlicht and H. Stuckenschmidt, “MapResolve,” in *Web Reasoning and Rule Systems – 5th International Conference, RR 2011, Galway, Ireland, August 29-30, 2011*, ser. Lecture Notes in Computer Science, vol. 6902. Springer, 2011, pp. 294–299.
- [53] Y. Ren, J. Z. Pan, and K. Lee, “Optimising parallel abox reasoning of EL ontologies,” in *Proceedings of the 2012 International Workshop on Description Logics, DL-2012, Rome, Italy, June 7-10, 2012*, ser. CEUR Workshop Proceedings, Y. Kazakov, D. Lembo, and F. Wolter, Eds., vol. 846. CEUR-WS.org, 2012. [Online]. Available: [http://ceur-ws.org/Vol-846/paper\\_61.pdf](http://ceur-ws.org/Vol-846/paper_61.pdf)
- [54] A. Fokoue, F. Meneguzzi, M. Sensoy, and J. Z. Pan, “Querying linked ontological data through distributed summarization,” in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, J. Hoffmann and B. Selman, Eds. AAAI Press, 2012. [Online]. Available: <http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5110>
- [55] G. Antoniou and F. van Harmelen, *A Semantic Web Primer, 2nd Edition*, 2nd ed. The MIT Press, March 2008.
- [56] J. Maluszynski and A. Szalas, “Living with Inconsistency and Taming Nonmonotonicity,” in *Datalog*, 2010, pp. 384–398.
- [57] 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.
- [58] M. J. Maher, “Propositional defeasible logic has linear complexity,” *CoRR*, vol. cs.AI/0405090, 2004.
- [59] D. Billington, “Defeasible logic is stable,” *J. Log. Comput.*, vol. 3, no. 4, pp. 379–400, 1993.
- [60] F. N. Afrati and J. D. Ullman, “Optimizing joins in a mapreduce environment,” in *In EDBT*, 2010.
- [61] K. B. R. Vernica, A. Balmin and V. Ercegovac, “Adaptive Mapreduce using Situation-Aware Mappers,” in *EDBT*.
- [62] 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>
- [63] F. Fische, “Investigation & Design for Rule-based Reasoning,” LarKC, Tech. Rep., 2010.

- [64] F. N. Afrati and J. D. Ullman, “Optimizing joins in a map-reduce environment,” in *EDBT*, 2010, pp. 99–110.
- [65] S. Liang, P. Fodor, H. Wan, and M. Kifer, “Openrulebench: an analysis of the performance of rule engines,” in *Proceedings of the 18th international conference on World wide web*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 601–610. [Online]. Available: <http://doi.acm.org/10.1145/1526709.1526790>
- [66] J. Nickolls and W. J. Dally, “The GPU computing era,” *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/MM.2010.41>
- [67] L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31, 2012*, ser. Frontiers in Artificial Intelligence and Applications, vol. 242. IOS Press, 2012.
- [68] A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, Eds., *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5823. Springer, 2009.