On Web-scale Reasoning

Jacopo Urbani

Copyright © 2013 by Jacopo Urbani

VRIJE UNIVERSITEIT

On Web-scale Reasoning

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op woensdag 9 januari 2013 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Jacopo Urbani

geboren te Arzignano, Italië

promotoren: prof.dr.ir. H.E. Bal
prof.dr. F.A.H. van Harmelen

“Time reveals the truth”

Seneca, 22 b.C.

Preface

After I completed my master, I had a constant feeling that my education was still not complete. What I needed was some more time to "think", to figure out what was missing, and I thought that staying in academia for few more years was a perfect way to find it out. Therefore, to me pursuing a PhD was simply the right thing to do even though - I must admit - I did not know very well what the next years would look like.

As the time was passing by, I began to understand more and more what I was doing. I started to grasp the essence of the work that is being done in the scientific community and I liked it more and more. I enjoyed very much to participate in the research process, and by working next to the most brilliant persons I have ever met I had the chance to experience how top-level research is currently conducted in one of the best universities in Europe.

In the last years I have learned several things, but what I believe is the most important one is that science is not only about *finding* some solutions (using the scientific method) but also about *sharing* them. I personally see science as a collaborative space (Wikipedia defines it as a "systematic enterprise") where people continuously exchange solutions to solve some common problems.

When I started my studies I had no idea on how to operate in such space. I did not know how to do research or on how to communicate it to other people. All of this was being taught to me either explicitly or by example. Therefore, since science is not only about doing research, but also about people, now is the time to express all my appreciation to everyone who helped me to learn this.

First of all, my gratitude goes to my supervisors, Henri Bal and Frank van Harmelen, who patiently guided and actively helped me in my research. Their teaching does not relate only to scientific matters. Through their example, they also showed me how anybody should do his job in the right way. I can truly say that they are two role models for me.

More in particular, I really admired in Henri his kindness and patience. I will always be grateful to him for being the one who really believed in me from the first moment. Even though I did my master thesis on a very different topic from the expertise of his group, he immediately offered me a job with a very high degree of freedom. I could not have had a better supervisor.

I also admire several qualities of my cosupervisor Frank, but there is one that stands above all, and it is his passion for science. With his enthusiasm, and his hard-working attitude he managed to transmit me one of the most important motivations that drive the work in academia: believing in what you are doing. Thanks very much for it.

Also, my sincere thanks to Spyros Kotoulas and Jason Maassen, who supervised me on a daily basis. They have always provided a concrete advice and help, and listened to my complaints on basically everything. I want to particularly mention Spyros not only for his precious help at work, but also for being a very nice friend outside the VU. Working with them was simply great, and I hope that I will have again the chance to do it again in the future.

I would also like to express my gratitude to all the coauthors of my publications, and to all the members of my PhD committee. Each of them played a significant role in shaping the content of this thesis, and because of this I thank them for their work and support.

In the last years, my colleagues in both Henri and Frank's groups have supported me very much, also on matters that were not related to my PhD. Ceriel Jacobs spent much time debugging and improving my code and endless conversations with my roommates Roelof, Timo, Nick, Alessandro, and all the others have certainly contributed to enlighten me on several issues in the Dutch culture. They are responsible for making my staying at the VU a very nice experience and I thank them very much for it.

Also, my apologies to all the users of the DAS-4 cluster are in order. Because of deadlines, I often abused the official policies and they patiently had to wait for my jobs to be finished. I hope it won't happen again.

Finally, outside my work I received a very important help from my own family, who has not only financially supported me through my studies, but also educated me for more than 28 years. Another big thanks goes to my wife's family, who has learned to endure my company and made the Netherlands my current home. To all of you, simply thanks for everything you did.

Last but not least, my respectful token of gratitude goes to my lady, Elsbeth, who patiently supported me for all the weekends where I was stuck working instead of going outside to enjoy the beautiful Dutch weather. Words cannot express what I feel for her. Now it's time for your PhD, baby :-).

One more thing. I mentioned that when I finished my master I felt that something was still missing. There is a famous Socrates' quote that says:

> *"The more I learn, the more I learn how little I know."*

Now that my academic education is over, I realized how painfully true this is.

Contents

1. Introduction **1**
1.1. Scope of research . . . 3
1.2. Summary of chapters . . . 5
1.3. Collaborations . . . 7

I Reasoning before query time **9**

2. Forward-chaining reasoning with MapReduce **11**
2.1. The MapReduce programming model . . . 12
2.1.1. A simple MapReduce example: term count . . . 13
2.1.2. Characteristics of MapReduce . . . 14
2.2. RDFS reasoning with MapReduce . . . 14
2.2.1. Example rule execution with MapReduce . . . 15
2.2.2. Problems of RDFS reasoning with MapReduce . . . 17
2.2.3. Loading schema triples in memory . . . 18
2.2.4. Data preprocessing to avoid duplicates . . . 19
2.2.5. Ordering the application of the RDFS rules . . . 20
2.3. OWL reasoning with MapReduce . . . 23
2.3.1. Challenges with OWL reasoning with MapReduce . . . 24

2.3.2. Limit duplicates when performing joins between instance triples . . . 26
2.3.3. Build sameAs table to avoid exponential derivation . . . 28
2.3.4. Perform redundant joins to avoid load balancing problems 30
2.4. Evaluation . . . 31
2.4.1. Implementation . . . 32
2.4.2. Experimental parameters . . . 32
2.4.3. Dataset and reasoning complexity . . . 35
2.4.4. Scalability . . . 36
2.4.5. Platform . . . 39
2.5. Related work . . . 40
2.6. Conclusion . . . 41

3. Distributed RDF data compression 45
3.1. Dictionary Encoding . . . 46
3.2. MapReduce Data compression . . . 48
3.2.1. Job 1: caching of popular terms . . . 50
3.2.2. Job 2: deconstruct statements, and assign IDs to terms 51
3.2.3. Job 3: reconstruct statements . . . 54
3.2.4. Storing the term IDs . . . 54
3.3. MapReduce data decompression . . . 55
3.3.1. Job 2: join with dictionary table . . . 56
3.3.2. Job 3: join with compressed input . . . 56
3.4. Evaluation . . . 57
3.4.1. Runtime . . . 58
3.4.2. Performance of the popular-term cache . . . 60
3.4.3. Scalability . . . 61
3.5. Related work . . . 64
3.6. Conclusions and Future Work . . . 65

4. Querying RDF data with Pig 67
4.1. SPARQL with Pig: overview . . . 69
4.1.1. Runtime query optimization . . . 70
4.1.2. Pig-aware cost estimation . . . 72
4.1.3. Dealing with Skew . . . 73
4.2. Evaluation . . . 77
4.2.1. Experiments . . . 78
4.3. Related Work . . . 83
4.4. Conclusions . . . 85

II Reasoning at query time 87

5. Hybrid-reasoning 89
5.1. Hybrid reasoning: Overview . . . 90
5.2. Hybrid Reasoning: Backward-chaining . . . 92
5.2.1. Our approach . . . 95
5.2.2. Exploiting the precomputation for efficient execution. . . 101
5.3. Hybrid Reasoning: Pre-Materialization . . . 102
5.3.1. Pre-Materialization algorithm . . . 102
5.3.2. Reasoning with Pre-Materialized Predicates . . . 104
5.4. Hybrid reasoning for OWL RL . . . 108
5.4.1. Detecting duplicate derivation in OWL RL . . . 111
5.5. Evaluation . . . 113
5.5.1. Performance of the pre-materialization algorithm . . . 113
5.5.2. Performance of the reasoning at query time . . . 115
5.5.3. Discussion . . . 120
5.6. Related Work . . . 121
5.7. Conclusions . . . 122

6. Reasoning and SPARQL on a distributed architecture 125
6.1. System architecture . . . 126
6.2. Data Storage . . . 127
6.3. Rule Execution . . . 129
6.4. SPARQL queries . . . 135
6.5. Evaluation . . . 138
6.5.1. Performance . . . 139
6.5.2. Scalability . . . 141
6.5.3. Efficiency . . . 143
6.6. Related Work . . . 144
6.7. Future Work and Conclusions . . . 146

III Discussion and conclusions 149

7. Conclusions: Towards a reasonable Web 151
7.1. 1st Law: Treat schema triples differently . . . 153
7.2. 2nd Law: Data skew dominates the data distribution . . . 154
7.3. 3rd Law: Certain problems only appear at a very large scale . . . 156
7.4. Conclusions . . . 158

IV Appendices 161

A. MapReduce Reasoning algorithms 163
A.1. RDFS MapReduce algorithms 163
A.2. OWL MapReduce algorithms 167

B. SPARQL queries 173
B.1. Queries for Yahoo! use-case 173
B.2. BSBM queries 174
B.3. LUBM queries 175

Bibliography 177

Chapter 1

Introduction

The Semantic Web [9] is an extension of the current World Wide Web, where the semantics of information can be interpreted by machines. Information is represented as a set of Resource Description Framework (RDF) statements [93], where each statement is made of three different terms: a *subject*, a *predicate*, and an *object*. An example statement is

```
<http://www.vu.nl> <rdf:type> <dbpedia:University>
```

This example states that the concept identified by the URI *http://www.vu.nl* is of type *dbpedia:University**. The Semantic Web is made of billions of such statements, which describe information on a very wide range of domains, from biomedical information [51] to government information [19]. URIs are often used to identify concepts to ensure unambiguity and to favor reuse in a distributed setting like the Web.

The RDF data model used in combination with ontology languages like OWL [52] allows generic applications to infer information that is not explicitly stated. For example, if we enrich our running example with the following triple:

```
<http://www.vu.nl> <rdfs:subClassOf> <foo:Public_Institution>
```

*Throughout this thesis we will often shorten URIs using prefixes for conciseness.

then we could infer that the concept identified by the URI `http://www.vu.nl` is also of type `Public_Institution`, even though this information is not explicitly stored.

This process, which is commonly referred to as *reasoning*, can be performed for a variety of purposes, like detecting inconsistencies or classifying data. In our case, we intend to use reasoning to enrich the results of queries by adding implicit information to the answer-set of the query.

When reasoning is applied to a large collection of RDF data, we label it as large-scale reasoning. When the input becomes even larger, so that its size can be compared the entire size of the Semantic Web, then the "large-scale" becomes "web-scale".

Web-scale reasoning is a crucial problem in the Semantic Web. In fact, at the beginning of 2009, the Semantic Web was estimated to contain about 4.4 billion triples[†]. One year later, the size of the Web had tripled to 13 billion triples and the current trend indicates that this growth rate has not changed.

With such growth, reasoning on a web scale becomes increasingly challenging, due to the large volume of data involved and to the complexity of the task. Most current reasoners are designed with a centralized architecture where the execution is carried out by a single machine. When the input size is on the order of billions of statements, the machine's hardware becomes the bottleneck. This is a limiting factor for performance and scalability.

A distributed approach to reasoning is potentially more scalable because its performance can be improved by adding more computational nodes. However, it is significantly more difficult to implement because it requires developing protocols and algorithms to efficiently share both data and computation. Therefore, the research question that we address in this thesis is:

> How can we perform reasoning to enrich query results over a very large amount of data (i.e. on a *web-scale*) using a parallel and distributed system?

To this end, we will present a number of algorithms that implement reasoning (and some other related tasks) on a parallel and distributed architecture. The results obtained in our evaluation show that a distributed approach is a viable option. In fact, by using our algorithms we were able to implement complex reasoning and querying on up to hundred billion triples, which accounts for about three times the entire size of the Semantic Web.

[†] `http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics`

Before presenting the main contribution of this thesis, in the next section we will define the scope of our research by making some ground assumptions and define more clearly our approach to solve our problem. After this, in Section 1.2, we will provide the outline of this thesis, briefly describing the content of each chapter to ease the understanding and navigation throughout this document. Finally, in Section 1.3 we will report an acknowledgment to external support that was given in conducting the research presented in this thesis.

1.1 Scope of research

As we state in our research question, we intend to apply reasoning to enrich the results of queries. Therefore, we exclude from our discussion other purposes for reasoning like detecting inconsistencies in the input data.

In this context we identify two main subtasks: the task of *reasoning* intended as the process to infer implicit information and the task of *querying* intended as the process to retrieve and return query results to the user. Depending on the type of reasoning that is performed, these two tasks can be either independent or strongly interlinked. While in this thesis we will mainly focus on the task of reasoning rather than of querying, we cannot ignore this last one because if the results of the reasoning cannot be used to enrich the queries then the entire process becomes meaningless.

Reasoning can be applied using a variety of methods. In this thesis we will focus only on reasoning that can be performed through the application of rules. More in particular, we will consider only monotonic rule-based reasoning. The motivation behind this choice lies on several considerations:

- in the Web, the data is distributed and it is difficult (or even impossible) to retract existing facts;
- there are some standardized rule sets (RDFS, OWL Horst, OWL 2 RL) that are widely used in the Semantic Web community.

We will consider two main approaches to perform rule-based reasoning: The first consists of applying all the rules before query-time, in order to derive all the implicit information. In this case rules are normally applied using a bottom-up strategy (also referred as *forward-chaining* or *materialization*) and then queries are answered using technologies developed in the field of data management. The second consists of applying the rules directly at query time, in order to limit the derivation to information that is related to the query.

Typically, this method is associated with a top-down application of the rules (also referred as *backward-chaining*).

Both approaches have advantages and disadvantages that make them suitable to different use cases. The main advantage of methods of the first type is that once all the derivation is computed no more reasoning is needed. A disadvantage is that such methods become inefficient if the user is interested in only a small portion of the input and the entire derivation is not needed.

Methods that perform reasoning directly at query time have the advantage that typically they do not require an expensive pre-computation and therefore are more suitable to datasets that change frequently. However, the computation required to execute the rules at query-time often becomes too expensive for interactive applications. Thus, it has until now been limited to either small datasets (usually in the context of expressive DL reasoners) or weak logics (RDFS inference).

In our context scalability is an important metric to evaluate our contribution. In principle, scalability can be evaluated according to two different criteria: *computational scalability* (i.e. the ability to perform more complex tasks) and *input scalability* (i.e. the ability to process a larger input). While it is essential that the reasoning process is *scalable* regarding both aspects, in our work we privilege the second type by sacrificing as little reasoning as necessary in order to handle a very large input.

Since scalability is very important in our context, in order to improve it we investigate the possibility to parallelize this process and to distribute it across several loosely coupled machines. Such an approach is potentially more scalable than a traditional sequential algorithm because it can scale on two dimensions: the hardware and the number of the machines. The downside of such choice is that while it might be beneficial in terms of performance, it introduces new challenges that must be properly addressed. These challenges can be grouped into three main classes of problems:

- **Large data transfers:** Reasoning is a data intensive problem and if the data is spread across many nodes, the communication can easily saturate the network or the disk bandwidth. Therefore, data transfers should be minimized;
- **Load balancing:** Load balancing is a very common problem in distributed environments. In the Semantic Web, it is even worse because data has a high skew, with some statements and terms being used much more frequently than others. Therefore, the nodes in which popular information is stored have to work much harder, creating a performance bottleneck;

- **Reasoning complexity:** Reasoning can be performed using a logic that has a worst-case complexity that can be exponential. The time it eventually takes to perform a reasoning task depends on both the considered logic and on the degree the input data exploits this logic. On a large scale, we need to find the best trade-off between logic complexity and performance, developing the best execution strategy for realistic datasets.

The reasoning methods that will be proposed in the next chapters will address these problems proposing some solutions that either solve or limit the effect of them on the overall performance. Therefore, we will frequently refer to these problems during the explanation to explain whether and how our solutions address them.

1.2 Summary of chapters

This thesis consists of seven chapters that are largely based from a collection of scientific papers that are either published or under submissions. In this section, we will outline the overall structure of this thesis and provide a brief description of each of these chapters pointing out which publications they are based on.

In general, the content of this thesis can be divided in three main parts. In the first part, we will focus on reasoning applied before query-time with the purpose of deriving all the possible inference. We call this part *Reasoning before query time* because we will apply the rules using a forward-chaining method before the user can query the data. This part is composed of three chapters:

- **Chapter 2**: We present a forward-chaining method that uses the MapReduce programming model to perform a materialization of all the derivation and present an evaluation using the DAS-4 computer cluster. The method presented in this chapter has won the third IEEE SCALE challenge [83] and it is extracted and adapted from the following publications:

    Scalable Distributed Reasoning using MapReduce. J. Urbani, S. Kotoulas, E. Oren, F. van Harmelen. In Proceedings of ISWC 2009.

    OWL reasoning with WebPIE: calculating the closure of 100 billion triples. J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. Bal. In Proceedings of ESWC 2010.

> *WebPIE: A Web-scale parallel inference engine using MapReduce.* J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. Bal. In Journal of Web Semantics, 10(0):59-75, 2012.

- **Chapter 3**: We describe a technique to compress the original RDF data using dictionary encoding in order to increase the performance of the reasoning. This compression technique is used in our reasoning algorithms to reduce the storage cost and to increase the performance of reasoning. The content of this chapter was previously published in the following publications:

  > *Massive Semantic Web data compression with MapReduce.* J. Urbani, J. Maassen, H. Bal. In Proceedings of the 1st MapReduce workshop at HPDC 2010.

  > *Scalable RDF data compression with MapReduce.* J. Urbani, J. Maassen, Niels Drost, Frank Seinstra, H. Bal. In Journal of Concurrency and Computation: Practice and Experience. John Wiley & Sons, Ltd. 2012.

- **Chapter 4**: We present a technique to execute SPARQL queries using the Pig language. Also in this case we use the MapReduce programming model to allow the execution of complex queries over a very large input. Therefore, this technique can be used after reasoning as presented in Chapter 2 to provide a complete environment where both reasoning and querying are possible. The content of this chapter was previously published in the following publication:

  > *Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig.* S. Kotoulas, J. Urbani, P. Boncz, P. Mika. In Proceedings of IWSC 2012.

In the second part of this thesis we will focus on the problem of performing reasoning at query-time, which is the second main approach that we consider. To this purpose we developed a technique that performs a small precomputation beforehand and applies the remaining at query-time using a top-down technique (also called backward-chaining). Because of this, we titled this part *Reasoning at query time.* This part is based on the work published in

> *QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases.* J. Urbani, F. van Harmelen, S. Schlobach, H. Bal. In Proceedings of ISWC 2011.

Since we report a much more detailed explanation of the content of this paper, we divided the presentation into two chapters in order to ease the presentation:

- **Chapter 5**: We describe the main idea of our technique and analyze fundamental properties like its soundness and completeness using the theory of deductive databases. The content of this chapter led to an additional publication which is currently under submission at:

  QueryPIE: Hybrid reasoning with the OWL RL rules. J. Urbani, R. Piro, F. van Harmelen, H. Bal. Currently under submission at the Semantic Web Journal. 2012.

- **Chapter 6**: Here we focus on the implementation of this technique in a distributed setting. We abstracted the execution of the rules according to some characteristics and integrated their execution into a simple SPARQL engine (This last feature is not described in our ISWC '11 publication).

The third and last part consists of a single chapter which is **Chapter 7**. In this chapter, we try to abstract the technical and low-level contribution presented in the previous chapters into some generic considerations that are behind the performance of our methods. This work aims to identify some key principles that can be used as a guideline to implement scalable reasoning on a distributed setting or as an advice for further research that addresses similar challenges in a different context.

1.3 Collaborations

A large part of the content of this thesis is extracted from a series of scientific publications that were written in collaboration with other researchers.

In this section we acknowledge their contribution that they have given to this work. More in particular, we mention the contribution of Peter Boncz and Peter Mika, who, next to providing for useful comments in the development process, have also ran the experiments using Virtuoso and the Yahoo! infrastructure. Also, we would like to acknowledge the major contribution that Robert Piro gave in formalizing and proving the correctness of the algorithms presented in Chapter 5. Finally, we would like to thank Ceriel Jacobs who has performed all the experiments presented in Chapter 6.

The work presented in this thesis was partially funded by the Dutch national research program COMMIT, and by the EU Project FP7-215535, The Large Knowledge Collider (LarKC).

Part I

Reasoning before query time

Chapter 2

Forward-chaining reasoning with MapReduce

In this chapter we will describe a MapReduce method to implement a forward-chaining reasoner to materialize all possible derivations using the RDFS and OWL $pD*$ rules.

The choice of MapReduce as programming model is motivated by the fact that MapReduce is designed to limit data exchange and alleviate load balancing problems by dynamically scheduling jobs on the available nodes. However, simply encoding the rules using MapReduce is not enough in terms of performance, and research is necessary to come up with efficient distributed algorithms.

There are several rulesets that apply reasoning with different levels of complexity. In this chapter, we will first focus on the RDFS [33] semantics, which has a ruleset with relatively low complexity. We propose three optimizations to address a set of challenges: ordering the rules to avoid fixpoint iteration, distributing the schema to improve load balancing and grouping the input according to the possible output to avoid duplicate derivations.

Second, in order to find the best tradeoff between complexity and performance, we extend our technique to deal with the more complex rules of the OWL ter Horst fragment [78]. The reason to choose this fragment is that semantics of RDFS is not expressive enough in some particular use cases [90] and the OWL ter Horst fragment is a de facto standard for scalable OWL

reasoning.

This fragment poses some additional challenges: performing joins between multiple instance triples and performing multiple joins per rule. We overcome these challenges by introducing three novel techniques to deal with a set of problematic rules, namely the ones concerning the `owl:transitiveProperty`, `owl:sameAs`, `owl:someValuesFrom`, and `owl:allValuesFrom` triples.

To evaluate our methods, we have implemented a prototype called WebPIE (Web-scale Parallel Inference Engine) using the Hadoop framework. We have deployed WebPIE on a 64-node cluster as well as on the Amazon cloud infrastructure and we have performed experiments using both real-world and synthetic benchmark data. The obtained results show that our approach can scale to a very large size, outperforming all published approaches, both in terms of throughput and input size. To the best of our knowledge it is the only approach that demonstrates complex Semantic Web reasoning for an input of 10^{11} triples.

This chapter is organized as follow: first, in Section 2.1, we give a brief introduction to the MapReduce programming model. This introduction is necessary to provide the reader with basic knowledge to understand the rest of the chapter.

Next, in Section 2.2, we focus on RDFS reasoning and we present a series of techniques to implement the RDFS ruleset using MapReduce. Next, in Section 2.3 we extend these technique to support the OWL ter Horst fragment (also referred as $pD*$ fragment). In Section 2.4 we provide the evaluation of WebPIE. Finally, the related work and the conclusions are reported in Sections 2.5 and 2.6 respectively.

The techniques are explained at a high level without going into the details of our MapReduce implementation. In Appendix A, we describe the implementation of the WebPIE implementation at a lower level providing the pseudocode of the most relevant reasoning algorithms.

2.1 The MapReduce programming model

MapReduce is a framework for parallel and distributed processing of batch jobs [21]. Each job consists of two phases: a map and a reduce. The mapping phase partitions the input data by associating each element with a key. The reduce phase processes each partition independently. All data is processed as a set of key/value pairs: the map function processes a key/value pair and produces a set of new key/value pairs; the reduce merges all intermediate values with the same key and outputs a new set of key/value pairs.

Algorithm 1 Counting term occurrences in RDF NTriples files

```
1  map(key, value):
2    // key: line number
3    // value: triple
4    emit(value.subject, blank); // emit a blank value, since
5    emit(value.predicate, blank); // only number of terms matters
6    emit(value.object, blank);
7
8  reduce(key, iterator values):
9    // key: triple term (URI or literal)
10   // values: list of irrelevant values for each term
11   int count=0;
12   for (value in values)
13     count++; // count number of values, equaling occurrences
14   emit(key, count);
```

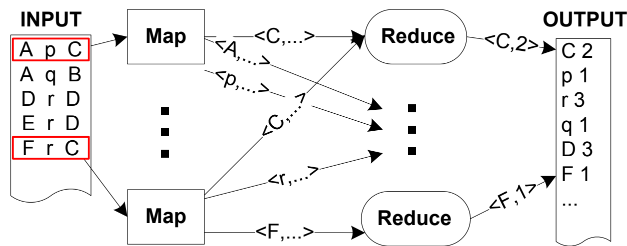


Figure 2.1: MapReduce processing

2.1.1 A simple MapReduce example: term count

We illustrate the use of MapReduce through an example application that counts the occurrences of each term in a collection of triples. As shown in Algorithm 1, the *map* function partitions these triples based on each term. Thus, it emits intermediate key/value pairs, using the triple terms (s,p,o) as keys and blank, irrelevant, value. The framework will group all intermediate pairs with the same key, and invoke the *reduce* function with the corresponding list of values, summing the number of values into an aggregate term count (one value was emitted for each term occurrence).

This job could be executed as shown in Figure 2.1. The input data is split in several blocks. Each computation node operates on one or more blocks, and performs the map function on that block. All intermediate values with the same key are sent to one node, where the reduce is applied.

2.1.2 Characteristics of MapReduce

This simple example illustrates some important elements of the MapReduce programming model:

- since the *map* operates on single pieces of data without dependencies, partitions can be created arbitrarily and can be scheduled in parallel across many nodes. In this example, the input triples can be split across nodes arbitrarily, since the computations on these triples (emitting the key/value pairs), are independent of each other.

- the *reduce* operates on an iterator of values because the set of values is typically far too large to fit in memory. This means that the reducer can only partially use correlations between these items while processing: it receives them as a stream instead of a set. In this example, operating on the stream is trivial, since the reducer simply increments the counter for each item.

- the *reduce* operates on all pieces of data that share the same key. By assigning proper keys to data items during the *map*, the data is partitioned for the *reduce* phase. A skewed partitioning (i.e. skewed key distribution) will lead to imbalances in the load of the compute nodes. If term x is relatively popular the node performing the *reduce* for term x will be slower than others. To use MapReduce efficiently, we must find balanced partitions of the data.

- Because the data resides on local nodes, and the physical location of data is known, the platform performs *locality-aware scheduling*: if possible, *map* and *reduce* are scheduled on the machine holding the relevant data, moving computation instead of data.

2.2 RDFS reasoning with MapReduce

The closure under the RDFS semantics [33] can be computed by applying all RDFS rules iteratively on the input until no new data is derived. Rules with one antecedent can be applied with a single pass on the data. Rules with two antecedents are more challenging to implement since they require a join over two parts of the data: if the join is successful, we derive a consequence.

It is well known that the RDFS closure is infinite, and we will describe how we ignore some of the RDFS rules which produce either trivial results or

1:	$s\ p\ o$ (if o is a literal)	$\Rightarrow$ _:n rdf:type rdfs:Literal
2:	p rdfs:domain x, $s\ p\ o$	$\Rightarrow s$ rdf:type x
3:	p rdfs:range x, $s\ p\ o$	$\Rightarrow o$ rdf:type x
4a:	$s\ p\ o$	$\Rightarrow s$ rdf:type rdfs:Resource
4b:	$s\ p\ o$	$\Rightarrow o$ rdf:type rdfs:Resource
5:	p rdfs:subPropertyOf q, q rdfs:subPropertyOf r	$\Rightarrow p$ rdfs:subPropertyOf r
6:	p rdf:type rdf:Property	$\Rightarrow p$ rdfs:subPropertyOf p
7:	$s\ p\ o$, p rdfs:subPropertyOf q	$\Rightarrow s\ q\ o$
8:	s rdf:type rdfs:Class	$\Rightarrow s$ rdfs:subClassOf rdfs:Resource
9:	s rdf:type x, x rdfs:subClassOf y	$\Rightarrow s$ rdf:type y
10:	s rdf:type rdfs:Class	$\Rightarrow s$ rdfs:subClassOf s
11:	x rdfs:subClassOf y, y rdfs:subClassOf z	$\Rightarrow x$ rdfs:subClassOf z
12:	p rdf:type rdfs:ContainerMembershipProperty	$\Rightarrow p$ rdfs:subPropertyOf rdfs:member
13:	o rdf:type rdfs:Datatype	$\Rightarrow o$ rdfs:subClassOf rdfs:Literal

Table 2.1: RDFS rules (D)[33]

ignore derivations that are considered bad style. For similar reasons, we do not add the RDFS axiomatic triples to the input data.

The rest of this section explains in detail how we can efficiently apply the RDFS rules with MapReduce. First, in Section 2.2.1, we describe how to implement an example rule in MapReduce in a straightforward way. Next, we describe the problems that arise with such a straightforward implementation and we discuss some optimizations that we introduce to solve, or at least reduce, such problems.

2.2.1 Example rule execution with MapReduce

Applying one rule means performing a join over some terms in the input triples. Let us consider rule 9 from Table 2.1, which derives `rdf:type` based on the sub-class hierarchy. This rule contains a single join on variable x, and it can be implemented with a *map* and *reduce* function, as shown in Figure 2.2 and Algorithm 2.

In the *map* function, we process each triple and output a key/value pair, using as value the original triple, and as key the triple's term (`s,p,o`) on which the join should be performed. To perform the join, triples with `rdf:type` should be grouped on their object (eg. "x"), while triples with `rdfs:subClassOf` should be grouped on their subject (also "x"). When all emitted tuples are

Algorithm 2 Naive sub-class reasoning (RDFS rule 9)

```
1  map(key, value):
2    // key: linenumber (irrelevant)
3    // value: triple
4    switch triple.predicate
5    case "rdf:type":
6      emit(triple.object, triple); // group (s rdf:type x) on x
7    case "rdfs:subClassOf":
8      emit(triple.subject, triple); // group (x rdfs:subClassOf y) on x
9
10 reduce(key, iterator values):
11   // key: triple term, eg x
12   // values: triples, eg (s type x), (x subClassOf y)
13   superclasses=empty;
14   types=empty;
15
16   // we iterate over triples
17   // if we find subClass statement, we remember the super-classes
18   // if we find a type statement, we remember the type
19   for (triple in values):
20     switch triple.predicate
21     case "rdfs:subClassOf":
22       superclasses.add(triple.object) // store y
23     case "rdf:type":
24       types.add(triple.subject) // store s
25
26    for (s in types):
27     for (y in classes):
28      emit(null, triple(s, "rdf:type", y));
```

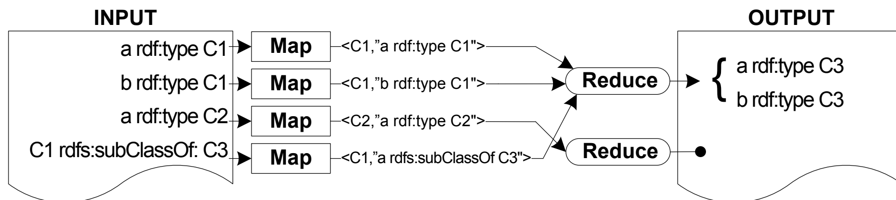


Figure 2.2: Encoding RDFS rule 9 in MapReduce.

grouped for the reduce phase, these two will group on "x" and the reducer will be able to perform the join.

One execution of these two functions will not find *all* corresponding conclusions because there could be some derived statements which trigger other joins. For example, in order to compute the transitive closure of a chain of n `rdfs:subClassOf`-inclusions, we would need to apply the above *map/reduce* steps $log_2 n$ times.

The algorithm we presented encodes only rule 9 of the RDFS rules. We would need to add other, similar, algorithms to implement each of the other rules. These other rules are interrelated: one rule can derive triples that can serve as input for another rule. For example, rule 2 derives `rdf:type` information from `rdfs:domain` statements. After applying that rule, we would need to re-apply our earlier rule 9 to derive possible superclasses. Thus, to produce the complete RDFS closure of the input data using this technique we need to add more *map/reduce* functions and keep executing them until we reach a fixpoint.

2.2.2 Problems of RDFS reasoning with MapReduce

The previously presented implementation is straightforward, but inefficient. The problems are:

Derivation of duplicates. We encoded, as example, only rule 9 and we launched a simulation over the Falcon dataset, which contains 35 million triples. After 40 minutes the program had not yet terminated, but had already generated more than 50 billion triples and filled our disk space. Considering that the unique derived triples from Falcon are no more than 1 billion, the ratio of unique derived triples to duplicates is at least 1:50. With such duplicates ratio, this implementation cannot scale to a large size because the communication and storage layers will fail to store the

additional data.

Join with schema triples. If we execute the joins as described in the example, there will be some groups which will be consistently larger than others and the system will be unable to efficiently parallelize the computation, since a group has to be processed by one single node.

Fixpoint iteration. In order to compute the closure, we need to keep applying the rules until we finish deriving new information. The number of iterations depends on the complexity of the input. Nevertheless, we can research whether a specific execution order leads to fewer iterations than another.

In the next sections, we introduce three optimizations that address these three problems to greatly decrease the time required for the computation of the closure. These optimizations are: (i) limit the number of duplicates with a pre-processing step; (ii) optimize the joins by loading the schema triples in memory; (iii) minimize the number of iterations needed by ordering the execution of rules according to their dependencies.

2.2.3 Loading schema triples in memory

When we perform the data joins required by the rule executions, we make a distinction between schema and instance triples. With schema triples we refer to those triples which have a RDFS or OWL term (except `owl:sameAs`) as predicate or object and that are used in the rules bodies. For example, `(:A rdfs:subClassOf :B)` is a schema triple while `(:a rdf:type :B)` is an example of instance triple.

Typically, schema triples are by far less numerous than instance triples [35]. Consider the Billion Triple Challenge dataset, which is a dataset of data crawled from the Web. This dataset was built to be a realistic representation of the Semantic Web and therefore can be used to infer statistics which are representative for the entire web of data. The results, shown in Table 2.2, confirm this assumption and allow us to exploit this fact to execute the joins more efficiently.

Before we continue our explanation, we must note that all the RDFS rules that require a join have at least one schema triple as antecedent. This means that all the RDFS joins are either between two schema triples or between one schema triple and one instance triple. We can exploit this fact to load all schema triples in memory and perform the join with the input triples in a streaming fashion.

Schema type	Number	Fraction
domain, range (p rdfs:domain D, p rdfs:range R)	30.000	0.004%
sub-property (a rdfs:subPropertyOf b)	70.000	0.009%
sub-class (a rdfs:subClassOf b)	2.000.000	0.2%

Table 2.2: Schema triples (number and fraction of total triples) in the Billion Triple Challenge dataset

As an illustration, let's consider rule 9 of Table 2.1: The set of `rdf:type` triples is typically far larger than the set of `rdfs:subClassOf` triples. Essentially, what we do is to load the small set of `rdfs:subClassOf` triples in memory and launch a MapReduce job that joins the many instance triples it receives as input (the `rdf:type` triples) with the in-memory schema triples. This methodology is different from the straightforward implementation where we were grouping the antecedents during the map phase and performing the join during the reduce phase (see the example in Section 2.2.1).

The advantages of performing a join against in-memory schema triples are: (i) We do not need to repartition our data in order to perform the join, meaning that we are reducing *data transfers*. (ii) For the same reason, the *load balance* is perfect, since any node may process any triple from the input. (iii) we can calculate the closure of the schema in-memory, avoiding the need to iterate $log_2 n$ times over our input, in practically all cases.

The main disadvantage of this method is that it works only if the schema is small enough to fit in memory. We have shown that this holds for generic web data but there might be some specific contexts in which this assumption does not hold anymore.

2.2.4 Data preprocessing to avoid duplicates

The join with the schema triples can be executed either during the map phase or during the reduce phase. If we execute it during the map phase, we can use the reduce phase to filter out the duplicates.

Let us take, as an example, rule 2 (`rdfs:domain`) of the RDFS ruleset. Assume we have an input with ten different triples that share the same subject and predicate but have a different object. If the predicate has a domain associated with it and we execute the join in the mappers, the framework will output a copy of the new triple for each of the ten triples in the input. These triples can be correctly filtered out by the reducer, but they will cause significant overhead because they will need to be stored locally and be transferred

over the network.

After initial experiments, we have concluded that the number of duplicates generated during the map phase was too high and it was affecting the overall performance. Therefore, we decided to move the execution of the join to the reduce phase and use the map phase to preprocess the triples.

To see how it works, let us go back to rule 2. Here, we can avoid the generation of duplicates if we first group the triples by subject and then execute the join over the single group. Grouping triples by value means that all derivations involving a given s will be made at the same reducer, making duplicate elimination trivial.

This methodology can be generalized for other joins by setting the parts of the input triples that are also used in the derived triple as the intermediate key and the part that should be matched against the schema as value. These parts depend on the applied rule. In the example above, the only part of the input that is also used in the output is the subject while the predicate is used as matching point. Therefore, we will set the subject as key and the predicate as value. Since the subject appears also in the derived triple, it is impossible that two different groups generate the same derived triple. Eventual duplicates that are generated within the group can be filtered out by the reduce function with the result that no duplicates can be generated.

This process does not avoid all duplicate derivations because we can still derive duplicates against the original input. Looking back at rule 2, when the reduce function derives that the triple's subject is an instance of the predicate's domain, it does not know whether this information was already explicit in the input. Therefore, we still need to filter out the derivation, but only against the input and not between the derivations from different nodes.

2.2.5 Ordering the application of the RDFS rules

We have analyzed the RDFS ruleset to understand which rule may be triggered by which other rule. By ordering the execution of rules according to their data dependencies, we can limit the number of iterations needed to reach full closure.

Rules 1, 4, 6, 8, 10 are ignored without loss of generality. These rules have one antecedent and can be implemented at any point in time with a single pass over the data and the outcome they produce cannot be used for further non-trivial derivation.

Also rules 12 and 13 have a single antecedent and therefore can be implemented with a single pass over the input. However, their outcome can be used

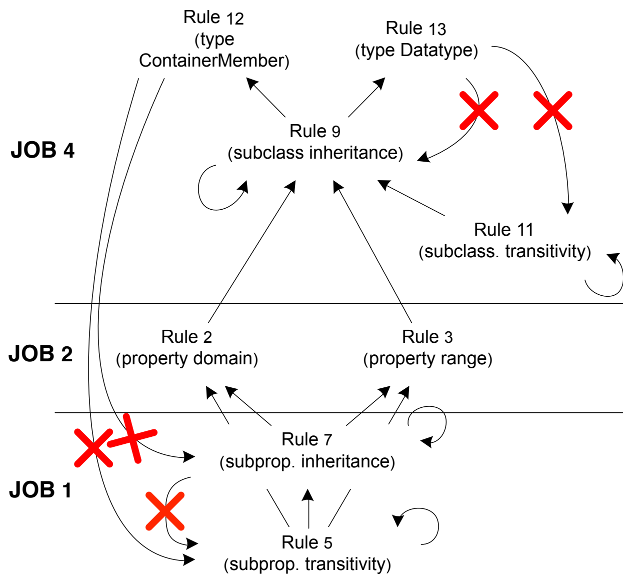


Figure 2.3: Relation between the various RDFS rules. The red cross indicates the relations that we do not consider.

for further non-trivial derivation and therefore we must include them in our discussion.

First, we have categorized the rules based on their output:

- rules 5 and 12 produce schema triples with *rdfs:subPropertyOf* as predicate;
- rules 11 and 13 produce schema triples with *rdfs:subClassOf* as predicate;
- rules 2, 3,and 9 produce instance triples with *rdf:type* as predicate;
- rule 7 may produce arbitrary triples.

We also have categorized the rules based on the predicates in their antecedents:

- rules 5 and 11 operate only on triples which have either *rdfs:subClassOf* or *rdfs:subPropertyOf* in the predicate position;
- rules 9, 12 and 13 operate on triples with *rdf:type*, *rdfs:subClassOf* or *rdfs:subPropertyOf* in the predicate position;
- rule 2, 3 and 7 can operate on arbitrary triples.

Figure 2.3 displays the relation between the RDFS rules based on their input and output (antecedents and consequents). Arrows signify data dependencies. An ideal execution should proceed bottom up: first apply the sub-property rules (rule 5 and 7), then rules 2 and 3, and finally rules 9, 11, 12 and 13.

Rules 12 and 13 can produce triples that serve the rules at the lower levels. However, looking carefully, we see that these connections are possible only in few cases: The output of rule 12 can be used to fire rules 5 and 7. In order to activate rule 5, there must be either a superproperty of `rdfs:member` or a subproperty of p. In order to activate rule 7 there must be some resources connected by p. We ignore the first case of rule 5, following the advice against "ontology hijacking" from [35] that suggests that users may not redefine the semantics of standard RDFS constructs. The other two cases are legitimate but we have noticed that, in our experiments, they never appear.

The same discussion applies to rule 13 which can potentially fire rules 9 and 11. Again, we have noticed that the legitimate cases when this happens are very rare.

The only possible loop we can encounter is when we extend the schema by introducing subproperties of `rdfs:subproperty`. In this case rule 7 could fire

rule 5, generating a loop. Although not disallowed, we have never encountered such case in our experiments.

Given these considerations, we can conclude that there is an efficient rule ordering which consistently reduces the number of times we need to apply the rules. The cases where we might have to reapply the rules either do not require expensive computation (for rules 12 and 13 an additional single pass over the data is enough) or occur very rarely or not at all.

Therefore, as our third optimization, we apply the rules as suggested in Figure 2.3, and launch an additional single-pass job to derive the possible derivation triggered by rules 12 and 13. Please notice that in this way we are performing *incomplete* RDFS reasoning since we intentionally exclude some cases that could potentially appear in the input and could lead to some additional derivation. The reason because we exclude some cases is that they would either produce undesirable inference (see the advice against "ontology hijacking") or that they occur only very rarely (or even not all) and they do not seem to be a real issue on real world datasets.

Nevertheless, if completeness is paramount, then it is cheap to verify after the rule execution whether one of these cases occur in the input or derivation. If these cases indeed occur, then a single application of the rules is not enough and we must repeat their execution until all the derivation is computed.

2.3 OWL reasoning with MapReduce

In the previous section, we have described how to perform efficient reasoning under the RDFS semantics. Here, we move to the more complex OWL semantics considering the ruleset of the ter Horst fragment [78]. The reasons of choosing this fragment are: (i) it is a *de facto* standard for scalable OWL reasoning, implemented by industrial strength triple stores such as OWLIM; (ii) it can be expressed by a set of rules; and (iii) it strikes a balance between the computationally infeasible OWL full and the limited expressiveness of RDFS. The OWL Horst ruleset consists of the RDFS rules [33], shown in Table 2.1 and the rules shown in Table 2.3.

As with the RDFS ruleset, we omit some rules with one antecedent (rules 5a, 5b). These rules can be parallelized efficiently and are commonly ignored by reasoners since they yield consequences that can be easily calculated at query-time. The other rules introduce new challenges.

In the remaining of this section we will describe these additional challenges and we will propose a new set of optimizations to address them. Again, we will defer thorough explanation of the algorithms to Appendix A.2.

1:	p rdf:type owl:FunctionalProperty, $u\ p\ v$, $u\ p\ w$	⇒	v owl:sameAs w
2:	p rdf:type owl:InverseFunctionalProperty, $v\ p\ u$, $w\ p\ u$	⇒	v owl:sameAs w
3:	p rdf:type owl:SymmetricProperty, $v\ p\ u$	⇒	$u\ p\ v$
4:	p rdf:type owl:TransitiveProperty, $u\ p\ w$, $w\ p\ v$	⇒	$u\ p\ v$
5a:	$u\ p\ v$	⇒	u owl:sameAs u
5b:	$u\ p\ v$	⇒	v owl:sameAs v
6:	v owl:sameAs w	⇒	w owl:sameAs v
7:	v owl:sameAs w, w owl:sameAs u	⇒	v owl:sameAs u
8a:	p owl:inverseOf q, $v\ p\ w$	⇒	$w\ q\ v$
8b:	p owl:inverseOf q, $v\ q\ w$	⇒	$w\ p\ v$
9:	v rdf:type owl:Class, v owl:sameAs w	⇒	v rdfs:subClassOf w
10:	p rdf:type owl:Property, p owl:sameAs q	⇒	p rdfs:subPropertyOf q
11:	$u\ p\ v$, u owl:sameAs x, v owl:sameAs y	⇒	$x\ p\ y$
12a:	v owl:equivalentClass w	⇒	v rdfs:subClassOf w
12b:	v owl:equivalentClass w	⇒	w rdfs:subClassOf v
12c:	v rdfs:subClassOf w, w rdfs:subClassOf v	⇒	v rdfs:equivalentClass w
13a:	v owl:equivalentProperty w	⇒	v rdfs:subPropertyOf w
13b:	v owl:equivalentProperty w	⇒	w rdfs:subPropertyOf v
13c:	v rdfs:subPropertyOf w, w rdfs:subPropertyOf v	⇒	v rdfs:equivalentProperty w
14a:	v owl:hasValue w, v owl:onProperty p, $u\ p\ w$	⇒	u rdf:type v
14b:	v owl:hasValue w, v owl:onProperty p, u rdf:type v	⇒	$u\ p\ w$
15:	v owl:someValuesFrom w, v owl:onProperty p, $u\ p\ x$, x rdf:type w	⇒	u rdf:type v
16:	v owl:allValuesFrom u, v owl:onProperty p, w rdf:type v, $w\ p\ x$	⇒	x rdf:type u

Table 2.3: OWL Horst rules

2.3.1 Challenges with OWL reasoning with MapReduce

Some of the rules in Table 2.3 can be efficiently implemented using the optimizations presented for the RDFS fragment. These are rules *3, 8a, 8b, 12a, 12b, 12c, 13a, 13b, 13c, 14a, 14b*.

The rest of the rules introduce new challenges:

Joins between multiple instance triples In RDFS, at most one antecedent can be matched by instance triples. In this fragment, rules 1, 2, 4, 7, 11, 15 and 16 contain two antecedents that can be matched by instance triples. Thus, loading one side of the join in memory (the schema triples) and processing instance triples in a streaming fashion no longer works because instance triples greatly outnumber schema triples and the main

memory of a compute node is not large enough to load the instance triples;

Exponential number of derivations This problem is evident with the *sameAs* derivations and we illustrate it with a small example. Consider that we have one resource which is used in n triples. When we add one synonym of this resource (which is also used in n triples), rule 11 would derive $2^1 \cdot n$ triples. If there are 3 synonyms, the reasoner will derive $2^3 \cdot n$ new resources and for l synonyms, it would derive $2^l \cdot n$. The I/O system can not sustain this exponential data growth, and soon enough, it would become a performance bottleneck;

Multiple joins per rule In RDFS, all the rules require at most one join between two antecedents. Here, rules 11, 15 and 16 require two joins. For example, rule 11 requires a join between `(u p v)` and `(u owl:sameAs x)` on `(u)` and a join between `(u p v)` and `(v owl:sameAs y)` on `(v)`;

Fixpoint iteration This problem was also present in the RDFS ruleset, but there, by making some assumptions, we could identify an execution order with no loops. This is not the case of OWL and we are required to iterate until fixpoint.

We propose three new optimizations that focus on the first three problems: (i) We limit the number of duplicates exploiting the characteristics of the rules. (ii) We build a synonyms table to avoid the materialization of the `sameAs` derivations. (iii) We execute part of the joins in memory when we have to perform multiple joins. In previous work [86] we ran some experiments changing the rule execution order to verify whether we could reduce the number of MapReduce jobs and therefore to improve the fourth problem of not having a fixpoint iteration. However, from our results we could not identify an optimal rule execution strategy that works in general and therefore in this chapter we will always consider a fixed rule execution order. Further investigations on this issue should be seen as future work.

In the remaining of this section we describe each optimization in more detail. These optimizations cannot be generically applied to any rule as it was the case with the RDFS fragment, but instead are tailored to the characteristics of individual rules or groups of rules in the ter Horst fragment.

2.3.2 Limit duplicates when performing joins between instance triples

In MapReduce, a join between instance triples can be performed only in the reduce phase, as shown in the example in Section 2.2.1. This execution can potentially lead to load balancing problems and exponential derivation of duplicates. In this section, we propose a series of techniques to avoid (or limit) the number of duplicate derivations.

The rules which require joins between instance triples are rules `1, 2, 4, 7, 11, 15` and `16`. We will defer discussion on the last four rules to Sections 2.3.3 and 2.3.4.

Rules 1 and 2 require a join between any two triples; however, if we look more carefully, we notice that the joins are on two resources, not on one, and the generic triples are filtered using information in the schema. In rule 1, the join is on the subject and predicate, while in rule 2 it is on the predicate and the object. Because we group on two resources, it is very unlikely that the reduce groups will be big enough to cause load balancing problems.

Rule 4 requires a three-way join between one schema triple and two instance triples. For readability we repeat the rule below:

```
if p rdf:type owl:TransitiveProperty
and u p w
and w p v
then u p v
```

At first sight, rule 4 seems similar to rules 1 and 2, suggesting that it can be implemented by partitioning triples according to `(pw)` (i.e. partition triples according to subject-predicate and predicate-object) and performing the join in-memory, together with the schema triple. However, there is a critical difference with rules 1 and 2: the descendant is likely to be used as an antecedent (i.e. we have chains of resources connected through a transitive relationship). Thus, this rule must be applied iteratively.

Applying rule 4 in a straightforward way will lead to a large number of duplicates, because every time the rule is applied, the same relationships will be inferred. For a transitive property chain of length n, a straightforward implementation will generate $O(n^3)$ copies while the maximum output only contains $O(n^2)$ unique pairs.

We can solve this problem if we constrain how triples are allowed to be combined. For example, consider the following three triples: `(a p b)`, `(b p c)` and `(c p d)` where `a`, `b`, and `c` are generic resources and `p` is a transitive property. Figure 2.4 graphically represents the corresponding RDF graph.

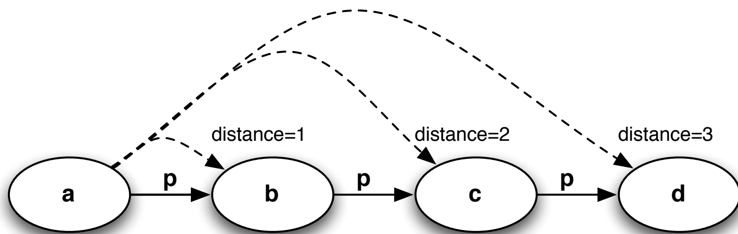


Figure 2.4: OWL transitive closure

Applying rule 4 consists of calculating the transitive closure of such chain; i.e. explicitly connect `a` with `c` and `d` and connect `b` with `d`.

With one MapReduce job we are able to perform a join only between triples with a shared resource. Therefore, we need log_n MapReduce jobs to compute the transitive closure of a chain of length n.

We define the *distance* between two subsequent resources in a transitive chain as the number of hops necessary to reach the second from the first one. In our example the terms `a` and `b` have distance one while the terms `a` and `c` have distance two because they are connected through `b`. When we execute the nth MapReduce job we would like not to derive triples which have a graph distance less than or equal to 2^{n-2} because these were already derived in the previous execution. We also would like to derive the new triples only once and not by different combinations. The conditions to assure this are:

- on the left side of the join (triples which have the key as object) we allow only triples with distance 2^{n-1} and distance 2^{n-2};

- on the right side of the join (triples which have the key as subject) we allow only triples with the distance greater than 2^{n-2}.

In the ideal case, we completely avoid duplicates. Nevertheless, when there are different chains that intersect, the algorithm will still produce duplicate derivations, but much fewer than without this optimization. This algorithm is described in pseudocode in Appendix A.2, Algorithm 23.

2.3.3 Build sameAs table to avoid exponential derivation

```
Rule 7                     Rule 11
-----------------------------------------------------------------
if v owl:sameAs w          if u p v
and w owl:sameAs u         and u owl:sameAs x
then v owl:sameAs u        and v owl:sameAs y
                           then x p y
```

To avoid exponential derivation, we do not directly materialize inferences based on `owl:sameAs` triples. Instead, we construct a table of all sets of resources connected by `owl:sameAs` relationships (rules 6, 7, 9, 10 and 11 from Table 2.3) and we always refer to the id of the sets. In other words, we choose a nominal representation for equivalent resources under `owl:sameAs`. This is common practice in industrial strength triple stores, and dealing with such a structure is much more efficient at query time. Note that this does not change the computational complexity of the task, since the `owl:sameAs` relationships are still calculated. The *sameAs* table simply provides a more compact representation, reducing the number of intermediate data that need to be processed and the size of the output.

This process makes rules 6, 9, 10 redundant, since the results of their application will be merged using the synonyms table.

Nevertheless, we still need to apply the logic of rule 7 and rule 11 (which are reported at the beginning of this section).

Both rules are problematic because they involve a two-way and a three-way join between instance triples. As before, and because the join is on instance data, we cannot load one side in memory. Instead we are obliged to perform the join by partitioning the input on the part of the antecedents involved in the join.

Again, such an approach would cause severe load balancing problems. For example, rule 11 involves joins on the subject or the object of an antecedent with no bound variables. As a result, the data will need to be partitioned on subject or object, which follow a very uneven distribution. This will obviously lead to serious load balancing issues because a single machine will be called to process a large portion of the triples in the system (e.g. consider the number of triples with `foaf:person` as object).

To avoid these issues, we first apply the logic of rule 7 to find all the groups of synonyms that are present in the datasets (i.e. resources connected by the `owl:sameAs` relation) and we assign a unique key to each of these groups. In other words, we calculate all non-singleton equivalence classes under

owl:sameAs. We store the pairs (resource, group_key) in a table that we call the sameAs table. Subsequently, we replace in our input all the occurrences of the resources in the sameAs table with their corresponding group key. In other words, we apply the same logic of rule 11, building a new dataset where we only refer to the single canonical representation for each equivalence class.

We will now describe how we can build the sameAs table and replace all items by their canonical representation using MapReduce.

Building the sameAs table

Our purpose is to group the resources connected with the sameAs relationship and assign a unique ID to each group.

For example, suppose that we have four triples: (a sameAs b), (b sameAs c), (c sameAs d) and (x sameAs y). These triples can be grouped in two groups: one containing a,b,c and the other containing x and y. We would like to assign a unique ID to each group and store this information in a table so that later on we can replace every instance of the resources (a, b, etc.) with the corresponding group ID.

Since the owl:sameAs relation is symmetric, the corresponding RDF graph of the sameAs triples is undirected ((a sameAs b) implies (b sameAs a)). What we do is to assign a unique ID to each resource and turn the undirected graph into a directed one by making the edges point to the node with higher ID. The node with the lowest ID will be the group's ID that will represent all the other nodes that are reachable from it.

Some nodes can be reached only through several hops. For example, suppose that a is the resource in our group with the lowest ID. The resource d can be reached only through b and c. In order to efficiently calculate all nodes reachable from the one with the lowest ID, we have developed a MapReduce algorithm (reported in pseudocode in the Appendix A, algorithm 24) where we incrementally create shorter paths starting from the node with the lowest ID removing all the intermediate edges that are no longer needed. The algorithm will terminate when all edges will originate from the node with the lowest ID and there will not be any intermediate one. Looking back at our example, the output of this algorithm would consist of the triples: (a sameAs b), (a sameAs c), (a sameAs d) and (x sameAs y). These triples will form the sameAs table since they contain the relation between each sameAs resource (e.g. a,b,c, etc.) and their corresponding group key (e.g. a).

Replacing resources with their canonical representation

Since our purpose is to replace in the original dataset the resources in the `sameAs` table with their canonical representation, we must perform a join between the input data and the information contained in the table. In principle, the join is executed by partitioning the dataset on the single term but such approach suffers from a severe load balancing problem since the term distribution is very uneven. We circumvent this problem by sampling the dataset to discover the most popular terms, and loading their eventual replacements in the memory of all nodes (similar to our technique for performing joins between schema and instance triples in RDFS). In our implementation, we typically sample a random subset of 1% of the dataset. When the nodes read the data in the input, they check whether the resource is already cached in memory. If it is, the nodes replace it on-the-fly and send the outcome to a random reduce task flagging it to be output immediately. For non-popular terms, the standard partitioning technique is used to perform the join, but because these terms are not popular, the partitioning will not cause load balancing issues.

Note that this approach is applicable to datasets with any popularity distribution: if we have a large proportion of terms that are significantly more popular than the rest, they will be spread to a large number of nodes, dissipating the load balancing issue. If there is a small proportion of popular terms, there will be enough memory to store the mappings.

2.3.4 Perform redundant joins to avoid load balancing problems

Rules 15 and 16 are challenging because they require multiple joins. Both rules are reported below:

```
Rule 15                           Rule 16
-----------------------------------------------------------------
if v owl:someValuesFrom w         if v owl:allValuesFrom u
and v owl:onProperty p            and v owl:onProperty p
and u p x                         and w rdf:type v
and x rdf:type w                  and w p x
then u rdf:type v                 then x rdf:type  u
```

In this section, we only discuss rule 15. The discussion for rule 16 is entirely analogous.

Rule 15 requires three joins on the antecedents' patterns. Two of them can be classified as schema patterns. These are the triples of the form (v

`owl:someValuesFrom` w) and (v `owl:onProperty` p). Since the schema is typically small, we can load both patterns in each node's main memory and perform the join between them using standard techniques.

Still, there are two joins left to perform. The first join will be between a small set (the result of the in-memory join) and a large one (the "type" triples). This join will produce a large set of triples that needs to be further joined with the rest of the input (the (u p x) pattern).

One option is to perform one of the remaining joins during the map phase and the other during the reduce. In such a scenario, the product of the first join is loaded in the nodes' main memory and the second join is performed against the "type" triples that are read in the input. When this join succeeds, the triples are forwarded to the reduce phase so that the third join against the remaining triples can be performed in the classical way.

There are two problems with this approach. First, it requires that one of the two instance patterns (either the "type" triples or the generic one) is always forwarded to the reduce phase to be matched during the third join. Second, the third join would be performed on a single resource and this will generate load balancing problems.

To solve these two problems, we perform the join between the schema and both instance patterns during the map phase. This means that the "type" triples will be matched on their objects (w) while all the other triples will be matched on the predicate (p). After that, we partition the triples not only on the common resource (x), but also on the common resource between the schema (v) that we can easily retrieve from the in-memory schema.

The advantages of this technique are: (i) we filter both instance patterns so that fewer triples will be forwarded to the reduce phase and (ii) we partition the triples on two resources instead of one, practically eliminating the load balancing issue.

The disadvantages are that we perform more joins than needed; however, the joins are performed against an in-memory data structure and therefore they do not introduce significant overhead in the computation.

2.4 Evaluation

In the previous sections we have described the problems of large scale reasoning and a number of optimizations as solutions to them. We have implemented a prototype called WebPIE and in this section we will evaluate its performance.

In our case performing an isolated evaluation of our optimizations is difficult because they cannot be evaluated independently to assess their real gain.

In fact, without these optimizations the system will fail due to the problems they address (excessive duplicates derivation, etc.) so that an individual evaluation of the optimizations is technically impossible. Since we are obliged to evaluate the system as a whole, we analyzed the performance varying the parameters that might influence the reasoning performance. For example, we tried to launch the reasoner on different inputs to verify how the performance is related to the input size. We also measured the scalability either increasing the input size or the number of nodes, etc. In the remaining of this section, we report a detailed description of the experiments we conducted.

This section is organized as follows. We start off by providing a description of the prototype we have developed. Then, we give an overview of the experimental parameters. Finally, we group our results and discuss them in sections 2.4.3-2.4.5.

2.4.1 Implementation

We have used the Hadoop 1.0.3 [31] framework, an open-source Java implementation of MapReduce, to implement WebPIE. Hadoop is designed for clusters of commodity machines and it can scale to clusters of thousand of machines. It uses a distributed file system built from the local disks of the participating machines, and manages execution details such as data transfer, job scheduling, and error management. We configured the cluster using seven mappers and reducers per nodes and activating the Google Snappy compression* to store intermediate results. The code used for our experiments is publicly available along with documentation and a tutorial [95].

2.4.2 Experimental parameters

The experimental parameters of our evaluation are described in this section. Table 2.4 shows an overview of these parameters, their range and the default values used. Unless otherwise specified, our experiments have been carried out using the default values. In the rest of this section, we briefly describe the meaning of these parameters.

Platform (Default value: DAS-4) We have performed most of our experiments on the DAS-4 VU cluster. In this cluster, each node is equipped with two quad-core Xeon processors, 24GB of main memory and two 1TB hard disk in RAID-0 mode. The nodes are interconnected through a Gigabit Ethernet.

*`http://code.google.com/p/snappy/`

Parameter	Range	Default value
Platform	DAS-4, Amazon EC2	DAS-4
Dataset	LDSR, LLD, Bio2RDF, LUBM	LDSR
Reasoning complexity	RDFS, OWL	OWL
No. nodes	1,2,4,8,16,32,64	32
Input size	1 billion-100 billion	1.3 billion

Table 2.4: Experimental parameters

We have also launched some experiments on the Amazon EC2 Cloud in order to facilitate comparison with future systems and to demonstrate the performance of WebPIE in the cloud. In this case, the Hadoop cluster consisted of 4 large Amazon instances, each with 7.5GB of main memory, 850GB of hard disk space and 4 EC2 CUs, roughly equivalent to 2 cores.

Datasets (Default value: LDSR) We launched WebPIE on a set of real-world datasets: The Linked Data Semantic Repository (LDSR [49]) dataset (later renamed to FactForge [24]) consists of several commonly-used datasets like DBPedia, Freebase and Geonames. The Linked Life Data (LLD) dataset [51] is a curated collection of datasets from the biomedical domain. In the same domain, the Bio2RDF dataset [12] is currently the largest dataset with Semantic Web data, consisting of more than 24 billion triples. Finally, we have also used the Lehigh University Benchmark (LUBM) [30], which is a benchmark for RDF, consisting of generated information in the academic domain. LUBM can generate arbitrarily large datasets keeping with reasoning complexity.

In Table 2.5, we report the number of statements, the number of statements for the OWL closure, and the number of distinct terms for each of these datasets. Also, for every dataset we report the supported reasoning complexity starting from one '+' if only limited RDFS/OWL reasoning is possible, and going to three '+' when almost all RDFS and OWL rules are supported. An intermediate number of '+' is to indicate the number of rules and should be used to get a rough indication of the reasoning complexity on that dataset.

Our system operates on compressed data, using the method presented in Chapter 3.

Reasoning complexity (Default value: OWL) We have performed experiments using both the RDFS ruleset (where the optimizations described

Dataset	Reasoning complexity	#Triples (Millions)	#Triples-OWL (Millions)	#Terms (Millions)
LDSR	+++	862	1790	259
LLD	++	694	1024	448
Bio2RDF	+	24000	25000	7302
LUBM	++	flexible	flexible	flexible

Table 2.5: Datasets used in our experiments (size, size under OWL closure and number of distinct terms)

Dataset	Input size	RDFS		OWL	
		Thr. Der.	Runtime	Thr. Der.	Runtime
LDSR	862 million	1284	621	189	6405
LLD	694 million	736	378	125	2651
Bio2RDF	24 billion	121	2703	23	26245
LUBM	1101 million	478	570	210	2362

Table 2.6: Reasoning time (in seconds) and throughput for only the derivation (measured in 1K triples/sec) per dataset.

in Section 2.2 apply) and the more expressive OWL-Horst ruleset (where the optimizations in Sections 2.2 and 2.3 apply). We have evaluated most of our optimizations using the OWL-Horst ruleset.

Number of nodes (Default value: 32) To measure the scalability of our approach given additional computational power, we launched the reasoner on a varying number of nodes.

Input size (Default value: 1.3B triples) Similarly, to measure the scalability of our approach regarding the input size we launched the reasoner varying the input size. We note that we cannot compare the performance of real-world datasets looking at their input size because they do not use the same language expressivity. For this reason, we will use the LUBM benchmark (which can scale arbitrarily in size) for evaluating the scalability of our system.

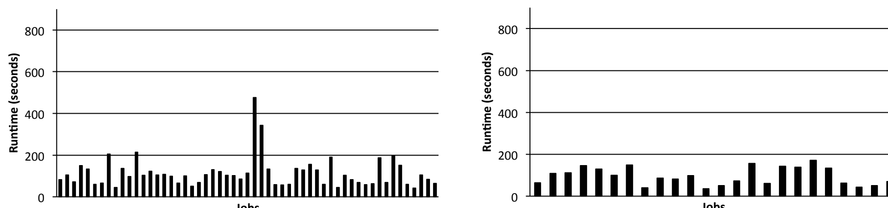


Figure 2.5: Runtime per job for LDSR (left) and LUBM (right)

2.4.3 Dataset and reasoning complexity

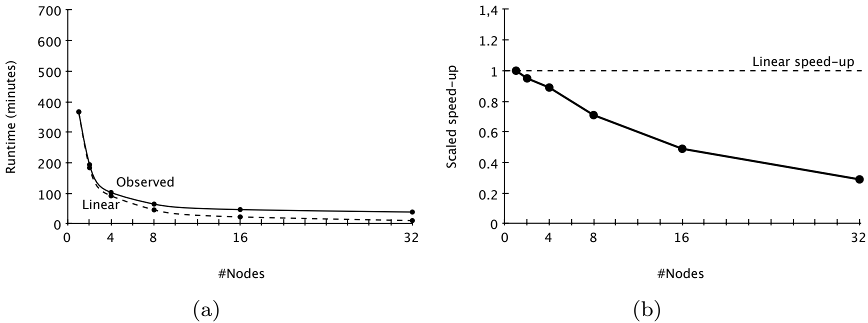
We launched WebPIE on different datasets and we report the runtimes in Table 2.6 for RDFS and OWL materialization. We make the following observations:

- RDFS reasoning is significantly faster than OWL reasoning, with a factor 2 to 7. This outcome was expected because RDFS has fewer and simpler rules than the OWL fragment and therefore it is easier to compute.

- The runtime and throughput of WebPIE is highly dependent on the logics employed by the input data (this is because different datasets trigger a different computation). Moreover, the computation time for the two logics is not strongly correlated: If we consider the RDFS fragment, we note that LLD yields the best results. However, for the OWL fragment, LUBM is the one with the best performance.

- For Bio2RDF, the ratio of derivations compared to the size of the input is low so that the majority of the execution consists of reading the input rather than inferring triples. This attests to the relatively weak inference possible over such large corpora.

To better illustrate how the input complexity affects the performance, we report in Figure 2.5 the runtime of every MapReduce job when WebPIE is launched on LUBM and LDSR. We observe the following: (i) reasoning with the LDSR dataset is significantly more complex. Namely, while we need a total



(a) (b)

Figure 2.6: (a) Runtime for the number of nodes (lower is better) (b) Scaled speed-up (higher is better)

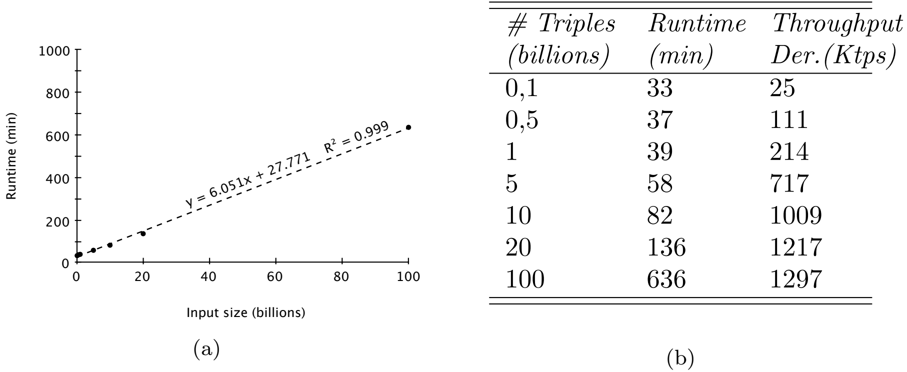
of 24 jobs for LUBM, while for LDSR, 55 jobs are required. (ii) The fastest jobs take around 30 seconds to finish. This is attributed to the platform overhead for starting a job. (ii) Most jobs take around between 30 and 60 seconds to finish. In practice, this means that the execution time is dominated by the platform overhead for starting jobs.

2.4.4 Scalability

We have evaluated the scalability of our approach in terms of input size and number of nodes. For these experiments, we have used the LUBM dataset, since to the best of our knowledge it is the only dataset that can be scaled to arbitrary size without any side-effect on its complexity.

Figure 2.6 summarizes our findings concerning the performance of our system for a varying number of nodes. Figure 2.6(a) shows the runtime for calculating the closure of LUBM together with the (theoretical) linear speed-up and Figure 2.6(b) shows the scaled speed-up. The latter is defined as $speed - up/no.nodes$, and is an indicator for the efficiency of our system, given additional computational nodes. A scaled speed-up of 1 means that given twice the number of nodes the system will perform twice as fast.

In these experiments, we note that indeed the performance of our system increases as we increase the number of nodes. However, such increase is not proportional to the added number of nodes and it eventually flattens out until there is almost no difference if we use more nodes. Such behavior is due to the fact that after the cluster reaches enough capacity to launch simultaneously



(a)

# Triples (billions)	*Runtime (min)*	*Throughput Der.(Ktps)*
0,1	33	25
0,5	37	111
1	39	214
5	58	717
10	82	1009
20	136	1217
100	636	1297

(b)

Figure 2.7: Runtime for increasing input size (on 64 nodes)

all the independent tasks of the jobs, it becomes irrelevant if more nodes are added. Therefore, we can conclude than regardless of the number of nodes, calculating the OWL closure of LUBM will not take less than approximately 40 minutes.

Figure 2.7 shows the runtime for an input size up to 100 billion triples on 64 nodes. Here, we see a similar situation as before: for small data sizes our throughput is reduced, since the runtime is dominated by platform overhead. However, as the size of the input grows, the throughput improves dramatically as the platform overhead is amortized over the longer runtime. Such results show that our method is more efficient for very large inputs where the platform overhead is not predominant as it is with smaller datasets.

In general we notice that the performance is approximatively linear with regard to the input size. This can be surprising considering that reasoning is a task with a computational complexity worse than linear. The reason behind the linear behavior is that LUBM is a benchmark that generates input with a reasoning complexity proportional to its size. That means that if we double the input size then also the computational complexity is doubled. Such feature of LUBM fits with the purpose of this experiment because in order to measure the scalability of this system with respect to the input size we do not want that additional complexity might influence the performance.

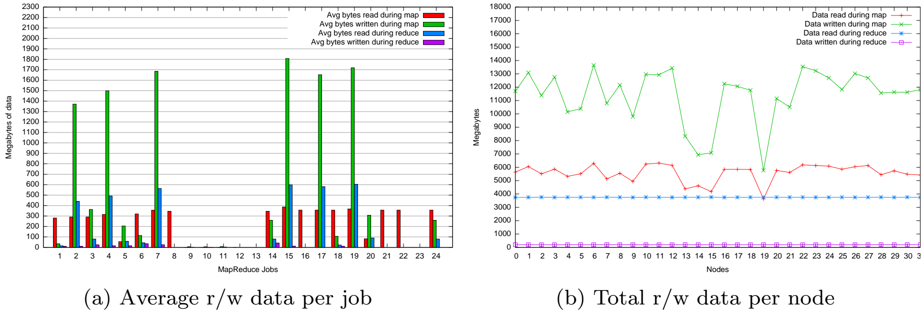
We have also analyzed the nodes' computational load during the reasoning task in order to understand if there are notable unbalances. To this purpose, we launched the reasoning process on the LUBM(8000) dataset using 32 DAS-4 machines and in Figure 2.8 we report the amount of data processed by the

nodes. In the first figure, we report the average amount of data that each node reads and writes during the map and reduce functions of the 24 MapReduce reasoning jobs. From the figure, we notice that the amount of intermediate data is considerably high on six cases. These jobs are the ones that clean the duplicates and such behavior is expected since they read and group the entire input in order to delete the duplicates. In the remaining cases, the amount of data read in input is higher than the amount effectively processed. This behavior indicates that in general only a small part of the input is relevant for reasoning.

In the second figure, we report the total amount of read and written data per node to verify that the computation is balanced across the nodes. Also in this case, we divided the total amount of data between the read and written in the map and reduce functions so that we can analyze their behavior independently. First we notice that the output of the map function is not equally distributed since there is a difference of a few gigabytes between the nodes. This behavior is explained as follows: in our prototype we split the triples in several files according to their predicate so that, depending on the reasoning rule, we can read a smaller amount of data skipping irrelevant data. For example, all the `rdf:type` triples are stored on files with extension "type". If there is a rule for which the "type" triples can't be used, we skipped reading them with consequent increase of performance. All the files are compressed and split in blocks of 64 megabytes. The blocks are distributed across the nodes and for each of them the node will apply the map function specific of the MapReduce job. Each time, the amount read is constant (64 megabytes) but the number of records might differ since the data is compressed and triples with the same predicate will take less space (due to compression optimizations). Therefore, some mappers might have to process more data and this explains the unbalance among them. We must note that this unbalance is not excessive since writing a few gigabytes of data does not take more than few minutes on a computation that is often more than a hour.

The amount of data processed during the reduce phase is uniformly spread across the nodes. This indicates that the optimizations that we explained in the previous sections do result in a good balancing with consequent benefit in terms of scalability.

From the above, it is safe to conclude that our approach scales gracefully both in terms of the number of nodes and data size, given than the reasoning task is large enough.



(a) Average r/w data per job (b) Total r/w data per node

Figure 2.8: Analysis of node computation per node during reasoning task

2.4.5 Platform

We have measured by launching the "top" command that, for the DAS-4, the jobs are essentially I/O bound. Thus, our method would benefit from a hardware setup with higher I/O throughput at the expense of lower CPU power. In our evaluation we activated data compression to store the intermediate map results with the attempt to reduce the I/O cost of transmitting and storing the intermediate data. While such choice makes the execution over small datasets a bit slower (because the data compression costs CPU time which is not needed if the data to write is small) its application over larger datasets is beneficial since the CPU time needed to compress the data is paid off by the advantage of storing much less data and this turns into a significant increase of performance.

On the other hand, for the Amazon EC2 Cloud, our jobs are mostly CPU bound. To facilitate comparison on a standard platform, we report that the execution time for LUBM (1 billion) on the four large instances on Amazon EC2 was 1064 minutes, 8 times slower than using 4 nodes on the DAS-4[†]. This degrade in performance is expected, considering that our experiments were running on virtualized and lower-spec hardware.

[†]This experiment was launched using an older version of the program which was producing one less MapReduce job due to a bug in the code.

2.5 Related work

In this section we describe the related work regarding scalable reasoning with high performance.

Hogan et al. [35, 36] compute the closure of an RDF graph using two passes over the data on a single machine. Initially they have implemented only a fragment of the OWL Horst semantics to allow more efficient materialization and to prevent "ontology hijacking". Later, the authors have extended their original approach to support a subset of OWL 2 RL and present an evaluation on 1.1 billion triples of a distributed implementation of their algorithms.

Schlicht and Stuckenschmidt [70] show peer-to-peer reasoning for the expressive ALC logic but focus on distribution rather than performance.

Soma and Prasanna [74] present a technique for parallel OWL inferencing through data partitioning. Experimental results show good speedup but only on very small datasets (1M triples) and runtime is not reported. In contrast, our approach needs no explicit partitioning phase and we show that it is scalable over increasing dataset size.

In [44] the authors present an inference engine that supports OWL 2 RL implemented inside the Oracle database. They describe a set of techniques to efficiently handle particular rules and they present an evaluation on a limited number of machines on datasets up to 13 billion triples. While they directly compare their performance against our work, we must point out that the two systems are quite different both in terms of hardware and software and therefore a direct comparison should be carefully weighted.

In [45, 64], the authors have presented a technique based on data-partitioning in a peer-to-peer network. A load-balanced auto-partitioning approach was used without upfront partitioning costs. Experimental results were however only reported for datasets of up to 200M triples.

In Weaver and Hendler [94], straightforward parallel RDFS reasoning on a cluster is presented. This approach replicates all schema triples to all processing nodes and distributes instance triples randomly. Each node calculates the closure of its partition using a conventional reasoner and the results are merged. To ensure that there are no dependencies between partitions, triples extending the RDFS schema are ignored. This approach is not extensible to richer logics, or complete RDFS reasoning, since, in these cases, splitting the input to independent partitions is impossible.

In Mutharaju et al. [54], a method for distributed reasoning with EL++ using MapReduce is presented, which is applicable to the EL fragment of OWL 2. This work was inspired by ours and tackles a more complicated logic than the one presented here. It is still an on-going work and no experimental

results are provided.

Newman et al. [60] decompose and merge RDF molecules using MapReduce and Hadoop. They perform SPARQL queries on the data but performance is reported over a dataset of limited size (70,000 triples). Husain et al. [38] report results for SPARQL querying using MapReduce for datasets up to 1.1 billion triples.

The work on Signal/Collect [76], introduces a new programming paradigm, targeted at handling graph data in a distributed manner. Although very promising, it is not comparable to our approach, since current experiments deal with much smaller graphs and are performed on a single machine.

Several proposed techniques are based on deterministic rendezvous-peers on top of distributed hashtables [8, 16, 25, 42]. However, because of load-balancing problems due to the data distributions, these approaches do not scale [45].

Some Semantic Web stores support reasoning and scale to tens of billions of triples [47]. We have shown inference on a triple set which is one order of magnitude *larger* than reported anywhere (100 billion triples against 12 billion triples). Furthermore, our inference is about 200 times *faster* (12 billion triples in 290 hours for LUBM against 10 billion triples in 1 hour and 22 minutes on 64 nodes) against the best performing reasoner (BigOWLIM). For LDSR, BigOWLIM 3.1 processes 14Ktriples per second [48] while our system yields a throughput of 189Ktps, on 32 nodes). It should be noted that the comparison of our system with RDF stores is not always meaningful, as our system does not support querying.

2.6 Conclusion

Summary In this chapter, we have shown a massively scalable technique for parallel RDFS and OWL Horst forward inference and demonstrated inference over 100 billion triples.

In order to improve the performance, we have introduced a number of key optimizations to handle a specific set of rules. These optimizations are:

- Load the schema triples in memory and, when possible, execute the join on-the-fly instead of in the reduce phase;
- Perform the joins during the reduce phase and use the map function to group the triples in order to avoid duplicates;
- Execute the RDFS rules in a specific order to reduce the number of MapReduce jobs;

– Limit duplicates when performing joins between instance triples using contextual information;

– Limit the exponential derivation of owl:sameAs triples building a sameAs table;

– Perform redundant joins to avoid load balancing problems.

Both in terms of processing throughput and maximum data size, our technique outperforms published approaches by a large margin.

Discussion of scope The computational worst-case complexity of even the RDFS/OWL Horst fragment precludes a solution that is efficient on all inputs. Any approach to efficient reasoning must make assumptions about the properties of realistic datasets, and optimize for those realistic cases. Some of the key assumptions behind our algorithms are: (a) The schema must be small enough to fit in main memory; (b) for rules with multiple joins, some of the joins must be performed in-memory, which could cause memory problems for some unrealistic datasets or for machines with very limited memory; (c) we assume that there is no ontology hijacking [35]; and (d) all the input is available locally in the distributed filesystem. The difference in performance on LLD, LDSR and LUBM shows that the complexity of the input data strongly affects performance. Although it is easy to create artificial datasets which would degrade the performance, we did not observe such cases in realistic data. In fact, the above assumptions (a)-(d) could also serve as guidelines in the design of ontologies and datasets, to ensure that they can be used effectively.

Discussion on MapReduce In this work, we have implemented a reasoning method using the MapReduce programming paradigm. In principle, the optimizations we have developed are not MapReduce-specific and can be applied on top of any infrastructure with similar workings. Originally, MapReduce operates in two discrete phases: a Map phase, where data is partitioned and a Reduce phase, where data in a given partition is processed. In some of our optimizations, we are deviating from this model by performing joins during the Map phase (effectively skipping the partitioning phase and operating on arbitrary parts of the input, together with a part of the input, replicated across all nodes).

On a larger scope, the Map and Reduce phases correspond to what a term partitioning system would do. In this sense, our system is similar to other systems doing term-partitioning (for example DHT-based systems). Replicating part of the input and performing a join in the Map phase would correspond

to a broadcast, in such a context. Thus, we expect that many of the optimizations presented in this chapter can be carried over to any system doing term partitioning. Our choice of MapReduce was mainly made for reasons of performance and scalability.

Future challenges The technique presented is optimized for the RDFS and OWL-Horst rules. Future work lies in reasoning over user-supplied rulesets, where the system would choose the correct implementation for each rule and the most efficient execution order, depending on the input.

Also, while we have identified a specific rule execution order for the RDFS rules, future work is needed to identify whether there exists a similar order to generally improve the performance for the OWL $pD*$ rules.

Furthermore, in general our approach cannot efficiently deal with incremental updates because every time the entire input must be read. We have performed some experiments to tackle this issue in previous work but the results still shows that such operation is in general expensive (even it can be improved considering only new triples in the derivation). Therefore, additional research is necessary to identify an efficient methodology to deal with this issue.

Finally, as with all scalable triple stores, our approach cannot efficiently deal with distributed data. Future work should extend our technique to deal with data streamed from remote locations.

Chapter 3

Distributed RDF data compression

Terms in RDF triples consist of long strings which can be either URIs or literals. Most Semantic Web applications, e.g. RDF storage engines, compress the statements to a more compact representation to save disk storage space and to increase performance. One of the most often used techniques to compress data is *dictionary encoding*. In dictionary encoding, each term in a data set is replaced with a numerical ID. Using the associated dictionary, data can be decompressed into its original uncompressed form. Because of its simplicity, this compressing technique is widely used in different fields.

Compressing the statements using a traditional centralized approach is becoming more and more time-consuming, and requires more and more memory because the amount of Semantic Web data is growing at an exponential rate (already comprising of many billions of statements [62]). To make dictionary encoding a feasible technique on a very large input, a distributed implementation is required. To the best of our knowledge, no distributed approach exists to solve this problem.

In this chapter we propose a technique to compress and decompress RDF statements using the MapReduce programming model [21]. Our approach uses a dictionary encoding technique that maintains the original structure of the triple and therefore applications can use this format natively. This technique can be used by all RDF applications that need to efficiently process a large

amount of data, such as RDF storage engines, network analysis tools, and reasoners.

In particular, this compression technique is a crucial component of our work on reasoners, because it allows us to reason directly on the compressed statements with a consequent increase of performance.

Characteristics of our compression technique are: (i) performance that scales linearly; (ii) the ability to build a very large dictionary of hundreds of millions of entries and (iii) the ability to handle load balancing issues with sampling and caching.

The remaining of this chapter is structured as follows. In Section 3.1 we discuss the conventional approach to dictionary encoding and highlight the problems that arise. Sections 3.2 and 3.3 describe how we have implemented the data compression and decompression in MapReduce. Section 3.4 evaluates our approach and Section 3.5 describes related work. Finally, we conclude and discuss future work in Section 3.6.

3.1 Dictionary Encoding

Dictionary encoding is often used because of it simplicity. In our case, dictionary encoding has also the additional advantage that the compressed data can still be manipulated by the application. Traditional techniques like gzip or bzip2 hide the original data so that reading without decompression is impossible.

We report in Algorithm 3 a simple sequential algorithm to compress and decompress RDF data using dictionary encoding. The algorithm starts by initializing the dictionary table. The table has two columns, one that contains the terms in their textual representation and one that contains the corresponding numerical ID. The algorithm reads all the statements and checks whether the terms exist in the table. If a term exists, the algorithm retrieves the numerical ID and replaces the term with the number. Otherwise, the algorithm assigns a new numerical ID, inserts a new pair in the table, and proceeds with the replacement. The algorithm outputs the compressed statements and the dictionary table.

Decompressing statements is even more straightforward than compression. For each term in the collection, the algorithm looks up the corresponding textual version in the dictionary table and replaces the numerical ID.

Unfortunately, a sequential implementation of dictionary encoding does not scale to the amounts of data present in the semantic web. One possible solution is to partition the input and process these partitions on several ma-

Algorithm 3 Sequential algorithm of dictionary encoding

```
1  compress_data(Iterator statements) {
2    dictionary_table.initialize()
3    for (statement in statements) {
4      for (term in statement) {
5        if (not dictionary_table.contains(term) {
6          newID = dictionary_table.get_new_ID()
7          dictionary_table.put(term, newID)
8          term = newID
9        } else {
10         //Replace the string term with a numerical ID
11         term= dictionary_table.getID(term)
12       }
13     }
14   }
15 }
16
17 decompress_data(Iterator statements) {
18   for (statement in statements) {
19     for (term in statement) {
20       textualTerm = dictionary_table.getText(term)
21       term = textualTerm
22     }
23   }
24 }
```

chines. In this scenario, a central server will store the dictionary table and all machines will query it to retrieve the numerical IDs. The problem with this approach is that all machines will constantly need to query the dictionary, making the dictionary server a performance bottleneck, and severely limiting the scalability of this approach.

Another potential problem when compressing RDF data is the size of the dictionary. For most applications of dictionary encoding the dictionary table is small enough to fit in the machine's main memory. For example, if we want to compress English text there are normally not more than a few hundred thousands distinct words in an English corpora. However, in a collection of billions of web statements there are normally hundreds of millions of unique terms. The dictionary for all these terms may not fit into the main memory of a single machine. In this context, even the distributed approach described above cannot handle this problem properly, as the central database machine will still need to hold the entire database. Because of its size, storing the entire dictionary on a single machine will lead to unacceptable performance.

To overcome the problems of runtime and memory usage, a completely new approach is required. In the next sections, we describe our approach which uses MapReduce to compress and decompress the data. Using this technique,

we are able to compress one billion of statements in less than two hours, more than one order of magnitude faster than conventional approaches.

3.2 MapReduce Data compression

MapReduce [21] is a distributed programming paradigm that we used in the previous chapter to implement scalable forward-chaining reasoning (a description of the framework is presented in Section 2.1).

We can use the same programming model to implement data compression. A naive implementation of dictionary encoding using MapReduce would read all terms in the dataset, group them by the same value, and replacing each value with a unique ID. This replacement step can be performed in a reduce function, so that we exploit the functionality offered by MapReduce to group all the pairs that share the same key.

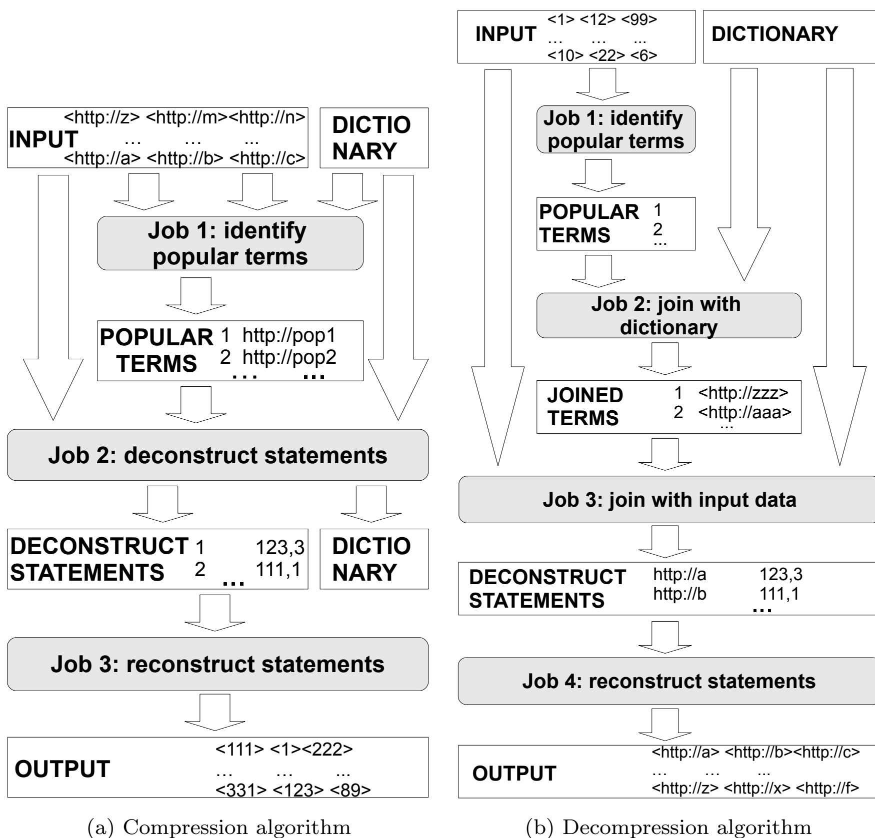
Unfortunately, such approach has three main issues that compromise its usage on a large scale. First, although we would like to process terms, our data is made up of statements (triples containing three terms). Therefore, a naive implementation would need to read the input three times, one to compress the subjects, one for the predicate and one for the object.

Secondly, such MapReduce implementation would still require a synchronized access to a central dictionary table to assign new numerical IDs and such solution would introduce a potential new performance bottleneck.

The third issue is represented by the fact that RDF data has a high skew. This means that a limited number of terms occurs many times in the dataset. Therefore, the list of all terms created at the reduce function may not fit in the main memory of a single machine. We cannot use a combiner function to alleviate this problem because such function works only on local data during the map phase, while the assignment process is performed during the reduce phase.

In our approach, we address these problems as follows. We tackle the first issue by first deconstructing all statements into a sequence of terms, compress these terms, then reconstruct the statements, now in a compressed form. In this way, we require to read the input only once.

We address the second issue by avoiding completely a centralized data structure and we gradually build the dictionary table in a distributed manner adding a new line every time a reducer processes a new group of terms. To avoid conflicts, the IDs number space is partitioned in as many partitions as the number of reducers and every reducer will be allowed to assign only numbers that are within its own partition.



(a) Compression algorithm (b) Decompression algorithm

Figure 3.1: Overall algorithms

We address the third issue by creating beforehand a list of all the *popular* terms , assign them an unique ID and perform the replacement directly on map function and not during the reduce as it is for all the other less popular terms.

Our initial implementation [85] created the dictionary table from scratch. As a result it was impossible to incrementally compress new data. Though this limitation makes the approach more efficient and faster, incremental updates is a feature which is strongly needed in real-world applications. In this chapter, we present an extended version of our initial algorithms which support

incremental updates, as well as other new features.

Figure 3.1a illustrates the overall compression algorithm of our approach. It consists of a sequence of three MapReduce jobs. The first job identifies the popular terms and assigns them a numerical ID. This algorithm is explained in Section 3.2.1. The second job (Section 3.2.2) deconstructs the statements, builds the dictionary table and replaces all terms with a corresponding numerical ID. The last job (Section 3.2.3) will read the numerical terms and reconstruct the statements in their compressed form. Finally, in Section 3.2.4 we explain how we can further compress the data by efficiently storing it on disk.

3.2.1 Job 1: caching of popular terms

In RDF web data, the distribution of the terms is highly skewed, having few popular terms and many that occur only few times. The purpose of the first job is to identify the most popular terms so that we can treat them in a different way. In order to do so, the job counts the occurrences of terms and creates a list of the most popular terms. Since the input is large, counting all the occurrences is an expensive operation. To increase performance, we randomly sample the input and extract the popular terms (i.e. the terms which appear more often than a specified threshold).

The algorithm is reported in pseudo code in Algorithm 4. It requires two parameters to run: *samplingPercentage* that sets the size of the sampling subset and *threshold* that sets the minimum occurrence to consider a term as popular.

In order to allow incremental updates, the input of the compression algorithm consists of both the data to compress, as well as any existing dictionary (in the code we use a wrapper data structure to differentiate these two types of data).

After the map function has sampled some terms, the reduce function checks if a popular entry has already an ID assigned in a previous update. If this is not the case, the function will assign it a number. As explained before, in order to avoid conflicts the function is allowed to assign only numbers from a specific range. To this purpose, in our prototype every reducer copies on the 13th bit of an integer it own task id and use the first 13 bits as internal counter. This means that there can be at most $2^{19} - 1$ reducers and each of them can assign not more than $2^{13} - 1$ numbers.

After compression is finished, we store the last numbers assigned to make the assignment process consistent during any further updates. If we do not do so, the same ID may be assigned to different terms during different executions.

Algorithm 4 Dictionary encoding: counting the terms occurrences

```
1   map(key, value) { /*key: void; value: one statement or one dictionary entry*/
2     if (key is number && value is text) {
3        emit(value, key)
4     } else { /*it's a statement*/
5        random = Random.newNumber(0:100)
6        if (random < samplingPercentage) {
7          emit(value.subject, null)
8          emit(value.predicate, null)
9          emit(value.object, null)
10       }
11    }
12  }
13
14  reduce(key, values) { /*key: one term in the collection; values: a sequence of null, or a
          number if it's a dictionary entry */
15    count = 0
16    number = 0;
17    for (value in values) {
18      if (value is not null) { /* The entry has already a number. Use it*/
19        number = value;
20      } else {
21        count += 1
22      }
23    }
24    if (count > threshold) {
25      if (number = 0)
26          number = assign_number(key)
27      emit(key,number)
28    }
29  }
```

3.2.2 Job 2: deconstruct statements, and assign IDs to terms

The purpose of this job is to deconstruct the statements and compress the terms with a numerical ID. This approach is used to avoid having to launch one job for each part of the statement.

The algorithm is shown in Algorithm 5. Before the map phase starts, every node reads the list of popular terms and corresponding ID from HDFS and loads them into the main memory. Since there are only few popular terms, they can fit in the main memory without any problem.

The map function reads the input that can be either a statement or a dictionary entry. If the input tuples are statements then the function assigns to each of them a numerical ID. Otherwise it forwards the dictionary entries to the reducers. Because the map tasks are executed in parallel, we partition the numerical range of the IDs so that each task is allowed to assign only a specific range of numbers to the statements. In our prototype we use a 64-bit number

Algorithm 5 Dictionary encoding: deconstruct the statements and assign IDs to terms

```
1  map(key, value) { /*key: void; value: one statement or one dictionary entry*/
2    statement_id = counter++
3    if (value is statement) {
4      for (term in value) {
5        if (popular_terms.contains(term)) {
6            id = popular_terms.getID(term)
7            emit(id, statement_id+term_position)
8        } else {
9           emit(term, statement_id+term_position)
10       }
11    }
12   } else { /*the value is a dictionary entry*/
13     emit(value, key)
14   }
15 }
16
17 reduce(key, values) { /*key: term; value: statements IDs + terms
18   position or corresponding numerical value*/
19   if (key is numeric) { /*The term was already replaced during the map*/
20     for(value in values)
21       emit(key,value)
22   } else { /*scroll the values to see whether there is already a
23     number*/
24     number = -1
25     storage.empty()
26     for (value in values) {
27       if (value is dictionary key)
28         number = value
29       else
30         storage.add(value)
31     }
32     if (number = -1) {
33       number = assign_number(key)
34       emit(number,key) /*Dictionary table entry*/
35     }
36     while (value in storage)
37       emit(number,value)
38   }
39 }
```

as identifier and we split it in two parts: the first four bytes will contain the id of the task that has processed the statement and the last four bytes will be used as incremental counter within the task. In this configuration, we can have at most $2^{32} - 1$ map tasks and each task can process an input of at most $2^{32} - 1$ statements. If we reserve more bytes for the internal map counter, then fewer map tasks can process a larger input, otherwise, if we reserve more bytes to store the map task number, then we can have more map tasks that process a smaller input.

For each term in the statements, the map function emits one intermediate pair. The key of the pair will be different whether the term is a popular one or not. If the term is a popular one, then the key will be the numerical ID retrieved from the in-memory cache that we have previously loaded. These pairs do not need to be further processed and can be output immediately. In case the term is not present in the cache, the map function will set the textual term as key. In both cases, the value of the pair will be the statement ID and the position of the term within the statement (1 if it is a subject, 2 if it is a predicate or 3 if it is an object). The statement IDs are used later to reconstruct each statement from its terms.

We use a specific partitioner function to assign the intermediate pairs to the reduce tasks. This function behaves as follows: if the key of the pair is a number, then the pair will be randomly assigned to a reduce task. Otherwise, if the key is a string, the hash function will be used to determine the reduce task. Using this partitioning technique all the pairs with a term that was not already replaced (i.e. those are the not popular) will be grouped together as usual. The other pairs do not require any further processing, therefore they are sent randomly to the nodes in order to avoid any load balancing issue.

During the reduce phase, each group will have either a textual or a numerical key. If the key is a number, the pairs in the group refer to an already converted popular term and the function will simply output the pairs. In the other case, the function proceeds assigning a numerical ID to the term.

To support incremental updates, we must first scroll through all the values to determine whether there is already a number assigned. The numerical IDs are assigned in a similar way as for the statements. We use a long number which is split in two parts: the first will contain the reduce task number (starting from 1, otherwise the numbers will conflict with the popular ones) while the second will be used as internal counter. For every pair in the group, the reduce function will output a corresponding pair with the numerical ID as key and the unchanged group input's value. The function will also emit an additional pair with the numerical value as key and the text as value. This pair will be stored as part of the dictionary table.

Therefore, the output of this job will be:

- A set of pairs which will have the numerical terms as keys and the information about the statements ID as values. These pairs will be used to reconstruct the original triples in the next job.

- A set of additional pairs with the numerical term as key and the corresponding textual representation as value. These pairs will form the dictionary table and be used to decompress the statements.

Algorithm 6 Dictionary encoding: reconstruct the statements

```
1  map(key, value) { /*key: numerical term; value: statement ID + term position*/
2    emit(value.statementID, key + value.term_position)
3  }
4
5  reduce(key, values) { /*key: statement ID; value: numerical term + term position*/
6    for (value in values) {
7      case (value.term_position) {
8        'subject' : statement.subject = value.term_id;
9                    break;
10       'predicate' : statement.predicate = value.term_id;
11                   break;
12       'object : statement.object = value.term_id;
13                   break;
14     }
15   }
16   emit(null, statement)
17 }
```

– The last values assigned to the URIs by the reducers to allow incremental compression if new data is added later.

3.2.3 Job 3: reconstruct statements

The last job reads in input what was returned by the previous one and reconstructs the statements using the numerical IDs. The algorithm is reported in Algorithm 6. The keys of the input pairs contain the terms in the numerical format. The values contain the statement ID and the position of the term within the statement (subject, predicate or object). The map function swaps the input key with the input value and emits a new pair which has as key the statement ID and as value the term plus its position.

The pairs will be grouped together for the reduce function. Since we now have set the statement ID as key, the pairs will be grouped according to the statement they belong to. Each group will have exactly three pairs, one for each part of the statement. For each group, the reduce function will read the three values and reconstruct the original statement with the numerical terms positioning them according to their position which is stored in the pairs values.

3.2.4 Storing the term IDs

The initial version of our prototype used a fixed size number of 8 bytes to store the term IDs. However, the full 8 bytes are not always required. For example, the popular terms requires only 4 bytes to be stored and since these are many

we can have a notable gain of performance if we use numbers with variable lengths.

As we explained in the previous sections, not-popular terms require 8 bytes because the first 4 bytes are used to store the task ID which has processed that term during the compression, while the remaining 4 bytes are used for the counter within this task. However, if there are only few reduce tasks, the first 4 bytes are too many because the task ID will take less space. For example, if we have 64 different reduce tasks, the algorithm will use at most 7 of the 32 reserved bits. The same holds for the last 4 bytes of the term ID. In case the number of statements is not very high, much of the space will be unused.

Simply reducing the number of bytes for the term ID is not an option, because eventual future updates may require the entire 8 bytes. Instead, we adaptively use fewer bytes for the term ID, increasing the space only when necessary.

In our solution, when we have to write a term ID on disk, we split the number in the two parts it is made of, and write each of them separately. Instead of using the fixed 4 bytes, we first write 2-3 bits with the number of bytes required to write the value, followed by the value itself. Thus, in case we have only 64 reduce tasks we will use only one byte to store the task ID and not four. The result of this optimization is that we further compress the data, reducing the space needed to store the values without losing any information.

3.3 MapReduce data decompression

After discussing the compression, we now focus on the decompression algorithm. It uses much of the same techniques as the compression. Again, first popular terms are handled separately, then the statements are deconstructed into terms, processed (in this case term IDs are replaced with their original value), and finally the statements are reconstructed using the textual terms instead of the numeric ones.

Figure 3.1b illustrates the overall decompression algorithm. It consists of a sequence of four MapReduce jobs. The first job identifies the popular terms. The second job performs the join between the popular resources and the dictionary table. The third algorithm deconstructs the statements and decompresses the terms performing a join on the input. The last job reconstructs the statements in the original format.

The first and the last jobs are analogous to the ones explained in sections 3.2.1 and 3.2.3 and therefore they will not be further explained. The others are described below.

Algorithm 7 Dictionary decoding: join with the popular terms

```
1  map(key, value) { /*key: term numerical ID; value: textual version of the term*/
2    if (popular_terms.contains(key)) {
3      emit(key,value)
4    }
5  }
```

3.3.1 Job 2: join with dictionary table

The purpose of this job is to retrieve the corresponding textual equivalents of the popular terms. Since the dictionary table contains hundreds of millions of entries, we need to launch a MapReduce job to retrieve them.

The algorithm is reported in Algorithm 7. The map function loads the popular terms in memory and reads the entries of the dictionary table. If the table entry matches one of the popular terms, then the function outputs the table entry. For this task a reduce function is not required.

3.3.2 Job 3: join with compressed input

The third job, which is reported in Algorithm 8, deconstructs the statements and performs the join with the dictionary table. Before the map function starts, a hash table is loaded in memory with the popular table dictionary entries calculated in the previous job. This hash table will have the popular textual terms as keys and the corresponding term IDs as values.

The input of the map function can be either a statement or a dictionary table entry. If the input record is a statement, then the map function deconstructs it as described in Section 3.2.2 and checks whether the terms are popular or not. If they are, the function will emit the pairs setting as key the text retrieved from the hash table. If not, the numerical ID will be used as a key and the join will be performed on the single term during the reduce phase. If the input record is an entry of the dictionary table, the function will emit a pair with the numerical ID as key and the text as value.

Similarly as section 3.2.2, we set a specific partitioner function to assign the pairs to the reduce tasks. The pairs with already converted terms will be randomly sent to the reducers and immediately returned. The other pairs will be grouped as usual and the join will be performed by the reduce function.

The reduce function stores the information on the statements in memory and saves the corresponding text of the numerical key in a variable. After this, the function will output new pairs with the textual term as keys and the values stored in memory as values. The last job will reconstruct the statements such

Algorithm 8 Dictionary decoding: join against all the terms

```
1  map(key, value) {
2    /*key: in case dictionary table entry this is the numerical ID, otherwise it is
         irrelevant*/
3    /*value: either a statement or the textual representation of the term*/
4
5    if (value is statement) {
6      statement_id = counter++
7      for (num_term in statement) {
8        if (popular_terms.contains(num_term)) {
9          textual_id = popular_terms.getID(num_term)
10         emit(textual_id, statement_id+term_position)
11       } else {
12         emit(num_term, statement_id+term_position)
13       }
14     }
15   } else {
16     emit(key, value)
17   }
18 }
19
20 reduce(key,values) {
21   if (key is text) { /*already processed popular term*/
22     for (value in values)
23       emit(key,value)
24   } else if (key is number) {
25      textual_term = null
26      for (value in values) {
27        if (value is statement) {
28          tmp_storage.add(value)
29        } else { /*value is the term textual repr.*/
30          textual_term = value
31        }
32      }
33      for (value in tmp_storage) emit(textual_term,value)
34   }
35 }
```

that the terms are restored to their original format and no longer encoded by numbers.

3.4 Evaluation

We implemented an open-source Java prototype of the presented algorithms using the Hadoop framework (version 0.20.2) to test their performance on a realistic scenario. The prototype was compiled and launched using Java 1.6.

To test our prototype, we set up our Hadoop framework on 32 nodes of our local DAS3 [18] cluster (the Hadoop and HDFS masters were running on the cluster's headnode). Each node is equipped with two dual-core 2.4 GHz AMD

Dataset (# stats.)	Input size (GB)		Comp. output size (GB)		Comp. rate	Runtime (sec.)		Throughput (stats/sec.)	
	Plain	GZip	Data	Dict.		Com.	Dec.	Com.	Dec.
DBPedia (110M)	17.4	1.9	1.4	1.0	7.33	665	619	166K	177K
Swoogle (78M)	16.2	1.3	1.0	0.6	10.60	424	516	186K	153K
LUBM (1101M)	158.9	7.0	14.0	1.9	10.02	4230	4847	260K	227K
Uniprot (1857M)	230.9	18.0	23.4	5.4	8.02	8577	8975	216K	207K
LDSR (1293M)	210.9	36.0	16.9	6.0	9.22	5685	6349	227K	204K
BTC (3180M)	672.8	27.0	38.7	4.3	15.64	13212	14573	240K	218K
LLD (694M)	113.0	3.4	8.2	3.0	10.06	3114	3612	223K	219K

Table 3.1: Execution time data compression and decompression on different datasets

Opteron CPUs, 4 GB of main memory and 250 GB of storage. The nodes were connected using Gigabit Ethernet. The Hadoop cluster was launched with the standard settings.

The input of our tests consists of a set of text files where every line contains one statement. The files were initially uploaded into the HDFS distributed filesystem and then processed by our MapReduce algorithms.

We evaluated the performance of our approach as follows. First, in Section 3.4.1, we focus on the runtime and we report some measurements performed on some real-world datasets. Next, in Section 3.4.2, we investigate the performance of the cache while last, in Section 3.4.3, we report the results of some tests to measure the scalability of our approach.

3.4.1 Runtime

We launched our prototype on several real-world and artificial datasets. DBPedia [20] is a collection of statements extracted from Wikipedia pages. Swoogle [77] contains statements collected by Web crawlers. LUBM [30] is a benchmark tool that is widely used in the community. LDSR [49] and BTC [11] are selected collection of datasets in the Web while Uniprot [81] and LLD [51] contain biological information.

These datasets differ from each other not only in terms of size but also

# Splits	Chunk size	Runtime (sec.)
1	99.4 GB	2531
2	49.5 GB	2789
4	24.5 GB	3188
8	12.4 GB	3753

Table 3.2: Execution time of incremental updates

because some contain larger statements than others (for example, in DBPedia there are statements that contain all the text of one Wikipedia page as object) and because they have a different term distribution. Therefore, it is interesting to see how our compression method works with different types of input.

The results are presented in Table 3.1. There, we report the size of the uncompressed input (both of the plain files and compressed with GZip), the size of the compressed output produced with our technique, the runtime of the compression and decompression algorithms, and other statistical information.

The size of the compressed output is the sum of the size of the data and of the dictionary table (this last one was further compressed using the standard zlib technique since typically applications do not need it until they have to decompress the data). We report the size of both data and dictionary table, to better understand the performance of the algorithms. The compression rate was calculated dividing the size of the uncompressed input (in plain files) by the total size of the compressed output, and varies from 1:7.33 (DBPedia) to 1:15 (BTC). We note that the compression rate increases when the dataset uses fewer terms, and consequently the dictionary table is smaller.

The throughput is slightly higher as the input size increases and therefore the platform overhead decreases. This is valid for the compression algorithm and to a lesser extent also for the decompression.

We also note that the compression algorithm is between 1% and 20% faster than the decompression algorithm for all but one dataset. We ran some tests to investigate the reason behind that and we observed that changing the settings of the cluster results in significant increase or decrease of performance. For example, if we set three mappers and three reducers per node instead of two, the decompression algorithm becomes faster on large datasets but worse on smaller. We intentionally set the cluster with the standard settings because we were interested on having a common base to evaluate the efficiency of the algorithm rather than measure the fastest absolute performance. However, we highlight the fact that by tuning the parameters, we can substantially change the runtime of the compression. This explains the small difference in

Threshold	# Cache	Runtime (sec.)	Speedup
Disabled	0	3526	1
100M	1	2387	1.48
50M	1	2387	1.48
10M	12	1991	1.77
5M	36	2039	1.73
1M	111	2019	1.75
0.5M	195	2053	1.72

Table 3.3: Cache speedup for the compression algorithm.

performance with the results presented in our previous work [85].

The new feature of incremental updates increases the algorithms complexity and therefore negatively affects the performance. We analyzed how much slower the compression algorithm would be if, instead of compressing all the data in one time, we would compress the data in little chunks.

To this purpose, we launched the following experiment: we took a dataset (LUBM) and we compressed it in one time. Next, we split the input in two chunks and we compressed it in two phases. We further split it in 4 and 8 chunks and compress one chunk at the time. The results are presented in Table 3.2. As we expected, compressing chunks of data is slower than compressing all the data in one time, as we have to reread the entire dictionary, and check if terms are already present in the dictionary. The decrease in performance is roughly proportional to the size of the chunk. To avoid a considerable loss in performance, the updates should be done in large chunks, instead of small ones. Nevertheless, the capability to handle updates may be invaluable for certain real-world applications, and this could justify even larger drops in performance.

3.4.2 Performance of the popular-term cache

We evaluated the beneficial effects of the popular-terms cache by launching* the compression algorithm on a fixed input (LDSR) and changing the size of the cache. First, we disabled the cache, then, we increased the cache size by decreasing the threshold used to identify the popular terms.

*For this experiments we used the previous version of Hadoop (0.19.1) and disabled the feature of allowing automatic updates as otherwise the algorithm would run out of memory when the cache is disabled.

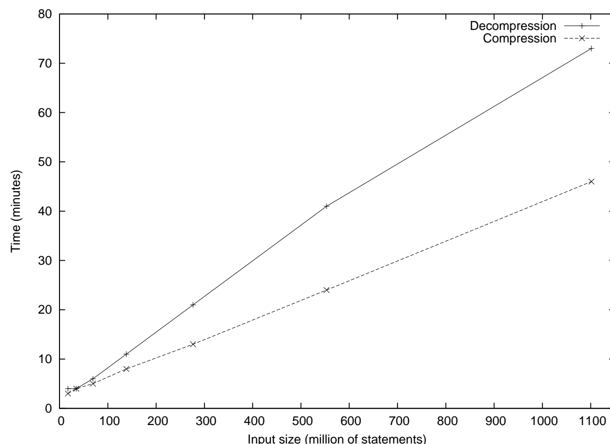


Figure 3.2: Scalability of the compression/decompression algorithms on the input size

The results are shown in Table 3.3. The first column shows the threshold we used to build the cache. The larger the threshold, the more occurrences of a certain term are required before we deem it *popular*. The second column gives the size of the cache for the popular terms. We immediately see that indeed the cache reduces the runtime, regardless of the threshold. If we lower the threshold then both the cache size and performance increase. This is valid up to a certain point: in fact, reducing the threshold below 10M no longer produces any notable performance difference. These results show that only a very small number of terms is responsible for the degradation of the performance (12 out of 259M) while the majority does not introduce any problem.

3.4.3 Scalability

We tested the scalability of our algorithm by launching the execution with different input sizes and varying the number of nodes. Since real-world datasets do not differ only in the size but also on other aspects (i.e. number of unique terms and size of the statements), we used the LUBM benchmark tool to generate artificial datasets of different sizes. LUBM can generate datasets with a proportional number of unique terms and with a fixed statement size, so we could test the performance without being influenced by the nature of the input.

We started by generating a dataset of 17 million statements. Then, we repeatedly doubled the size until 1.1 billion statements were generated. On each dataset we launched the compression algorithm, immediately followed by the decompression algorithm. The runtime is reported in Figure 3.2.

If we compare the runtime of the compression algorithm with the one of the decompression algorithm, we note that the latter becomes slightly slower as the input increases. This difference shows that the compression algorithm has a better scalability than the decompression algorithm regarding the input size. Such behavior could be explained that compressing a dataset generates less intermediate data than decompressing it since the algorithm performs the replacement as soon as possible. Therefore, while in the first case the amount of data could still fit in memory, in the second case it will require writing it on disk with consequent loss of performance.

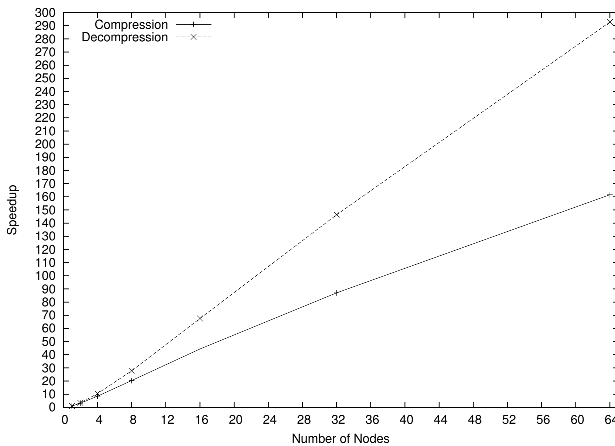
We also tested the scalability of our approach by launching the execution on a fixed input on clusters with a different number of nodes. We have used a LUBM dataset of 500 million statements as input and we kept the number of mappers (256) and reducers (128) constant. We varied the number of nodes, starting at 1 and doubling up to 64. In Table 3.3a we report the runtime of the two algorithms, the absolute speedup and the relative speedup, which is the speedup calculated compared to the previous line in the table (half the number of nodes). From the table, we see that the runtime decreases as we increase the number of nodes.

We have plotted the speedup in Figure 3.3b and we observe from it that it is superlinear in both cases. However, by looking at the relative speedup we notice that on larger numbers of nodes the performance gain decreases until it shows linear scalability. In this case, the superlinear behavior might be misleading because it implies better performance on a limited number of nodes, instead of more. However, the reason for such behavior can be explained from the fact that the Hadoop framework is designed to work on large clusters so that resources are more efficiently utilized in a distributed environment rather than on a single machine. For example, since in our case the performance is I/O bounded, if the computation is not spread between enough nodes, the performance degrades considerably (e.g. because of the too many requests a single disk has to process) which means that a single machine execution performs worse than it should when compared with two or more machines. This is why the decompression algorithm, which produces more I/O than the compression, has an initial higher speedup than the compression one.

Because of this, it is more appropriate to consider the execution time on a large cluster and, in doing that, we conclude that the two algorithms exhibit a linear speedup rather than a superlinear one.

Nodes	Runtime compr. (sec.)	Runtime decompr. (sec.)	Speedup compr.	Speedup decompr.	Relative speedup compr.	Relative speedup decompr.
1	2262	7027	-	-	-	-
2	727	2059	3.11	3.41	3.11	3.41
4	267	673	8.47	10.44	2.72	3.06
8	111	253	20.38	27.77	2.41	2.66
16	51	104	44.35	67.57	2.18	2.43
32	26	48	87.00	146.40	1.96	2.17
64	14	24	161.57	292.79	1.86	2.00

(a)



(b)

Figure 3.3: Scalability of the compression/decompression algorithms changing the number of nodes

3.5 Related work

Single machine dictionary encoding is used in RDF storage engines like Hexastore, 3Store and Sesame to store the information more efficiently [1, 15, 96]. Inference engines like WebPIE [82, 84, 86] and OWLIM [43] also compress the data with this technique.

An overview of compression techniques that can be applied to RDF data is presented in [26]. In this short paper, the authors consider three different techniques: the standard "gzip" compression, a compression based on adjacency lists and dictionary encoding. From their evaluation, they show that the compression highly depends on the structure of the data and that dictionary encoding is the most efficient if there is a considerable number of URIs in the dataset.

In [50] the authors propose a technique to apply structured dictionary encoding to the URIs present in the RDF files. This technique works at two levels: first it compresses the namespace of the URI and after it proceeds compressing the rest of the URI using the namespace as reference. Though the work is still in its early stages, the evaluation shows how indeed such compression technique is an improvement over the traditional gzip of between 19.5 and 39.5%.

Another work on the topic of compressing URLs is described in [53]. Here, the authors focus on the problem of maintaining efficient web caching and they present a simple URL table compression algorithm. The algorithm is based on hierarchical decomposition of the URL in order to aggregate common prefixes and use an incremental hashing function to minimize the collisions between the prefixes. By introducing this technique, the authors minimize the time needed to access the cache and compress the information at the same time.

Dictionary encoding is not only used within the Semantic Web but also in several other domains. In [98] dictionary encoding is used for image compression. In [55] the authors present some parallel techniques to compress data using an pre-existing dictionary.

In some domains, the dictionary is small enough to be kept in main memory. A good comparison between the performance of different in-memory data structures is given in [99]. From the comparison, it is clear that the hash table is the fastest data structure. [34] proposes a new data structure, called burst trie which maintains the strings in sorted, or near-sorted order, and has performance comparable to the one of a tree.

The decompression algorithm requires that we perform a join on the data but the original MapReduce paradigm does not provide any tool to perform efficient joins. An extension to the MapReduce programming model is called

Map-Reduce-Merge [97] and it aims to extend MapReduce to support data joins. Other frameworks, like Pig [63], or Hive [79] are built on top of Hadoop and they provide SQL-like languages to run queries on very large datasets.

3.6 Conclusions and Future Work

In this chapter we have proposed a technique to compress RDF data using the MapReduce programming model. We have shown how we can exploit the features of MapReduce to compress a large amount of data building a dictionary of hundreds of millions of entries.

We have implemented a prototype using the Hadoop framework. We evaluated the performance measuring the runtime using existing datasets and tested the scalability increasing the input size and number of nodes. The evaluation showed that the MapReduce algorithms are able to compress and decompress a large input with a high throughput and that both algorithms are more efficient for larger inputs. We also noticed that the compression algorithm is slightly more efficient than the decompressing algorithm.

The work we have presented can be extended in a number of ways. For example, these algorithms could be extended to work in different domains. Our approach is targeted at domains where there are large dictionaries but it could be interesting, for example, to see whether it could be adapted to perform generic text compression encoding the words with numbers. In this case, we could assign progressive IDs to the sentences and deconstruct them as we did with the statements.

Another interesting direction could be to build a structured dictionary where we could assign similar numbers to URIs which are similar as well (for example because they share the same namespace). Actually we assign numbers based on the hashcode of the URIs but if we can encode a notion of “closeness” or a similar semantical relationship in our process, all the applications which apply semantical operations, for example ranking of statements, would not need to decompress the numbers but use them directly with a consequent increase of performance.

Chapter 4

Querying RDF data with Pig

After the inference is computed, the user must face the problem of efficiently querying a large collection of data. Because of the large input size, the user queries can be translated into "killer" SPARQL queries that overwhelm the current state-of-the-art in RDF database systems. Such problems typically come down to formulating joins that produce huge results, or to RDF database systems that calculate the wrong join order such that the intermediate results get too large to be processed.

In this chapter we consider the problem of performing complex SPARQL queries on a very large knowledge base and propose a system that uses MapReduce and the Pig Latin language to perform an efficient execution. In doing so, we are considering some key issues:

Schema-less. A SPARQL query optimizer typically lacks all schema knowledge that a relational system has available, making this task more challenging. In a relational system, schema knowledge can be exploited by keeping statistics, such as table cardinalities and histograms that capture the value and frequency distributions of relational columns. RDF database systems, on the other hand, cannot assume any schema and store all RDF triples in a table with Subject, Property, Object columns (S,P,O). Both relational column projections as well as foreign key joins map in the SPARQL equivalent into self-join patterns. Therefore, if a query is expressed in both SQL and

SPARQL, on the physical algebra level, the plan generated from SPARQL will typically have many more self-joins than the relational plan has joins. Because of the high complexity of join order optimization as a function of the number of joins, SPARQL query optimization is more challenging than SQL query optimization.

Correlations. In order to estimate the join hit ratios with the same precision as relational systems, RDF systems would need to keep detailed statistics. Such detailed statistical information is simply not available, because of the sheer amount of possibly relevant selection criteria. Some systems keep statistics on certain "often used" traversal paths [59], while others try to guess hit ratios from the shape of the join patterns [57]. These methods, however, do not offer a general solution to this problem, and optimization of complex queries can lead to very bad plans even in state-of-the-art RDF systems.

RDF systems typically try to mitigate some of the problems by creating multiple permutations of the S,P,O columns, storing them in multiple B-tree indices (sometimes even all six possible permutations [10, 59, 96]). Such redundant storage is useful to efficiently answer leaf patterns in queries (in which two of the three S,P,O values are bound) and can also be used to estimate the size of such selections at query optimization time. However, such redundant storage can help only in the first join step of query execution.

MapReduce and Skew. Linked Open Data ETL tasks which involve cleaning, interlinking and inferencing have a high computational cost, which motivates our choice for a MapReduce approach.

In a MapReduce-based system, data is represented in files that can come from recent Web crawls. Hence, we have an initial situation *without* statistics and *without* any B-trees, let alone multiple B-trees. One particular problem in raw Web data is the high skew in join keys in RDF data. Certain subjects and properties are often re-used (most notorious are RDF Schema properties) which lead to joins where certain key-values will be very frequent. These keys do not only lead to large intermediate results, but can also cause one machine to get overloaded in a join job and hence run out of memory (and automatic job restarts provided by Hadoop will fail again). This is indeed more general than joins: in the sort phase of MapReduce, a large amount of data might need to be sorted on disk, severely degrading performance.

SPARQL on Pig. In this chapter, we describe a system that scalably executes SPARQL queries using the Pig Latin language [63] and we demonstrate its usage on a synthetic benchmark and on crawled Web data. For this task, we use a standard cluster and the large MapReduce infrastructure provided by Yahoo!.

The Pig Latin language provides operators to scan data files on a Hadoop cluster that form tuple streams, and further select, aggregate, join and sort such streams, both using ready-to-go Pig relational operators as well as using user-defined functions (UDFs). Each MapReduce job materializes the intermediate results in files that are replicated in the distributed filesystem. Such materialization and replication make the system robust, such that the jobs of failing machines can be restarted on other machines without causing a query failure. However, from the query processing point of view, this materialization is a source of inefficiency [75]. The Pig framework attempts to improve this situation by compiling Pig queries into a minimal number of Hadoop jobs, effectively combining more operators in a single MapReduce operation. An efficient query optimization strategy must be aware of it and each query processing step should minimize the number of Hadoop jobs.

Our work addresses these challenges and proposes an efficient translation of some crucial operators into Pig Latin, namely joins, making them robust enough to deal with the issues typical of large data collections.

Contributions. We can summarize the contributions of this work as follows. *(i)* We have created a system that can compute complex SPARQL queries on huge RDF datasets. *(ii)* We present a runtime query optimization framework that is optimized for Pig in that it aims at minimizing the number of MapReduce jobs, therefore reducing query latency. *(iii)* We describe a skew-resistant join method that can be used when the runtime query optimization discovers the risk for a skewed join distribution that may lead to structural machine overlap in the MapReduce cluster. *(iv)* We evaluate the system on a standard cluster and a Yahoo! Hadoop cluster of over 3500 machines using synthetic benchmark data, as well as real Web crawl data.

Outline. The rest of this chapter is structured as follows. In Section 4.1 we present our approach and describe some of its crucial parts. In Section 4.2 we evaluate our approach on both synthetic and real-world data. In Section 4.3 we report on related work. Finally, in Section 4.4, we draw conclusions and discuss future work.

4.1 SPARQL with Pig: overview

In this section, we present a set of techniques to allow efficient querying over data on Web-scale, using MapReduce. We have chosen to translate the SPARQL 1.1 algebra to Pig Latin instead of making a direct translation to a physical algebra in order to readily exploit optimizations in the Pig engine.

While this work is the first attempt to encode full SPARQL 1.1 in Pig, a complete description of such process is elaborate and goes beyond the scope of this chapter.

The remaining of this section is organized as follows: in Sections 4.1.1 and 4.1.2, we present a method for runtime query optimization and query cost calculation suitable for a batch processing environment like MapReduce. Finally, in Section 4.1.3, we present a skew detection method and a specialized join predicate suitable for parallel joins under heavy skew, frequent on Linked Data corpora.

4.1.1 Runtime query optimization

In this work, we adapt the ROX query optimization algorithm [2, 41] to SPARQL and MapReduce. ROX interleaves query execution with joins sampling, in order to improve result set estimates. Our specific context differs to that of ROX in that:

- SPARQL queries generate a large number of joins, which often have a multi-star shape [27].
- The overhead of starting MapReduce jobs in order to perform sampling is significant. The start-up latency for *any* MapReduce job lies within tens of seconds and minutes.
- Given the highly parallel nature of the environment, executing several queries at the same time has relatively small impact on the execution time of each query.

Algorithm 9 outlines the basic block of our join order optimization algorithm. To cope with the potentially large number of join predicates in SPARQL queries, we draw from dynamic programming and dynamic query optimization techniques, constructing the plans bottom-up and partially executing them.

Initially, we extract from the dataset the bindings for all statement patterns in the query and calculate their cardinalities. From initial experiments, given the highly parallel nature of MapReduce, we have concluded that the cost of this operation is amortized over the execution of the query since we are avoiding several scans over the entire input. Then, we perform a series of *construct-prune-sample* cycles. The construct phase generates new solutions from the partial solutions in the previous cycles. These are then pruned according to their estimated cost. The remaining ones are sampled and/or partially executed. The pruning and sampling phases are optional. We will

Algorithm 9 Runtime optimization

```
1   J: Set of join operators in the query
2   L: List of sets of (partial) query plans
3   void optimize joins(J) {
4    execute(J)
5    L_0:=(J)
6    i:=1
7    while (L_{i-1} ≠ ∅)
8     for (j ∈ L_{i-1})
9      for (I ∈ {L_0...L_{i-1}})
10      for (k ∈ I)
11       if (j ≠ k)
12        L_i.add(construct(j,k))
13     if (stddev(cost(L_i))/
14                mean(cost(L_i)) > t)
15      prune(L_i)
16      sample(L_i)
17     i:=i + 1
18  }
```

only sample if $stddev(costs)/mean(costs)$ is higher than some given threshold, so as to avoid additional optimization overhead if the cost estimates for the candidate plans are not significantly different.

Construct During the construct phase (lines 8-12 in Algorithm 9), the results of the previous iterations are combined to generate new candidate (partial) plans. A new plan is generated by either adding an argument to an existing node when possible (e.g. making a 2-way join a 3-way join) or by creating a new join node.

Prune We pick the $k\%$ cheapest plans from the previous phase, using the cost calculation mechanism described in Section 4.1.2. The remaining plans are discarded.

Sample To improve the accuracy of the estimation, we fully execute the plan up to depth 1 (i.e. the entire plan minus the upper-most join). Then, we use Algorithm 10 to perform bi-focal sampling.

There is a number of salient features in our join optimization algorithm:

- There is a degree of speculation, since we are sampling only after constructing and pruning the plans. We do not select plans based on their calculated cost using sampling, but we are selecting plans based on the cost of their 'sub-plans' and the operator that will be applied.

- Nevertheless, our algorithm will not get 'trapped' into an expensive join,

since we only fully execute a join after we have sampled it in a previous cycle.

- Since we are evaluating multiple partial solutions at the same time, it is essential to re-use existing results for our cost estimations and to avoid unnecessary computation. Since the execution of Pig scripts and our run-time optimization algorithm often materialize intermediate results anyway, the latter are re-used whenever possible.

4.1.2 Pig-aware cost estimation

Using a MapReduce-based infrastructure gives rise to new challenges in cost estimation. First, queries are executed in batch and there is significant overhead in starting new batches. Second, within batches, there is no opportunity for sideways information passing [58], due to constraints in the programming model. Third, when executing queries on thousands of cores, load-balancing becomes very important, often outweighing the cost for calculating intermediate results. Fourth, random access to data is either not available or very slow since there are no data indexes. On the other hand, reading large portions of the input is relatively cheap, since it is an embarrassingly parallel operation.

In this context, we have developed a model based on the cost of the following: Writing a tuple (w); Reading a tuple (r); The cost of a join per tuple. In Hadoop, a join can be performed either during the reduce phase (j_r), essentially a combination of a hash-join between machines and a merge-join on each machine, or during the map phase (j_m) by loading one side in the memory of all nodes, essentially a hash-join. Obviously, the latter is much faster than the former, since it does not require repartitioning of the data on one side or sorting, exhibits good load-balancing properties, and requires that the input is read and written only once; The depth of the join tree(d), when considering only the reduce-phase joins. This is roughly proportional to the number of MapReduce jobs required to execute the plan. Considering the significant overhead of executing a job, we consider this separately from reading and writing tuples.

The final cost for a query plan is calculated as the weighted sum of the above, with indicatory weights being 3 for w, 1 for r, 10 for j_r, 1 for j_m and a value proportional to the size of the input for d.

Algorithm 10 Bi-focal sampling in Pig

```
1  DEFINE bifocal_sampling(L, R, s, t)
2   RETURNS FC {
3  LS = SAMPLE L s;
4  RS = SAMPLE R s;
5  LSG = GROUP LS BY S;
6  RSG = GROUP RS BY S;
7  LSC = FOREACH LSG GENERATE flatten(group), COUNT(LSG) as c;
8  RSC = FOREACH RSG GENERATE flatten(group), COUNT(RSG) as c;
9  LSC = FOREACH LSC GENERATE group::S as S ,c as c;
10 RSC = FOREACH RSC GENERATE group::S as S ,c as c;
11 SPLIT LSC INTO LSCP IF c>=t, LSCNP IF c<t;
12 SPLIT RSC INTO RSCP IF c>=t, RSCNP IF c<t;
13
14 // Dense
15 DJ = JOIN LSCP BY S, RSCP BY S using 'replicated';
16 DJ = FOREACH RA GENERATE LSCP::c as c1, RSCP::c as c2;
17 // Left sparse
18 RA = JOIN RSC BY S, LSCNP BY S;
19 RA = FOREACH RA GENERATE LSCNP::c as c1, RSC::c as c2;
20
21 // Right sparseLA = JOIN LSC BY S, RSCNP BY S;
22 LA = FOREACH LA GENERATE LSC::c as c1, RSCNP::c as c2;
23 // Union results
24 AC = UNION ONSCHEMA DA, RA, LA;
25
26 $FC = FOREACH AC GENERATE c1*c2 as c;
27 }
```

4.1.3 Dealing with Skew

The significant skew in the term distribution of RDF data has been recognized as a major barrier to efficient parallelization [45]. In this section, we are presenting a method to detect skew and a method for load-balanced joins in Pig.

Detecting skew

To detect skew (and estimate result set size), we are presenting an implementation of bi-focal sampling [28] for Pig and report the pseudocode in Algorithm 10. Similar to join optimization, one of the main goals is to minimize the number of jobs. L, R,s and t refer to the left and right sides of the join, the sampling rate and the number of tuples that the memory can hold respectively. Initially, we sample the input (lines 3-4), group by the join keys (lines 5-6) and

count the number of occurrences of each key (lines 7-10). We split each side of the join by key popularity using a fixed threshold, which is dependent on the amount or memory available to each processing node (lines 11-12). We then perform a join between tuples with popular keys (lines 15-16) and a join for each side for tuples with non-popular keys and the entire input (lines 18-24).

This algorithm generates seven MapReduce jobs out of which two are Map-only jobs that can be executed in parallel and four are jobs with Reduce phases that happen concurrently in pairs. In fact, it is possible to implement our algorithm in two jobs, programming directly on Hadoop instead of using Pig primitives.

Determining join implementation

In Pig, it is up to the developer to choose the join implementation. In our system, we choose according to the following:

- If all join arguments but one fit in memory, then we perform a replicated join. Replicated joins are performed on Map side by loading all arguments except for one into main memory and streaming through the remaining one.
- If we have a join with more than two arguments and more than one of them do not fit in memory, we are performing a standard (hash) join.
- If the input arguments or the results of the (sampled) join present significant skew, we perform the skew resistant join described in the following section.

Skew-resistant join

As a by-product of the bi-focal sampling technique presented previously, we have the term distribution for each side of the join and an estimate of the result size for each term. Using this information, we can estimate the skew as the ratio of the maximum number of results for any key to the average number of results over all keys. Hadoop has some built-in resistance to skewed joins by means of rescheduling jobs to idle nodes, which is sufficient for cases where some jobs are slightly slower than others. Furthermore Pig has a specialized join predicate to handle a skewed join key popularity [29], by virtue of calculating a key popularity histogram and distributing the jobs according to this. Nevertheless, neither of these algorithms can effectively handle skewed joins where a very small number of keys dominates the join. We should further note

Algorithm 11 Skew-resistant join

```
1  DEFINE skew_resistant_join(A, B, k)
2   RETURNS result {
3  SA = SAMPLE A 0.01; //Sample first side
4  GA = GROUP SA BY O;
5  GA2 = FOREACH GA GENERATE COUNT_STAR(SA), group;
6  OrderedA = ORDER GA2 BY $0 DESC;
7  PopularA = LIMIT OrderedA k;
8  SB = SAMPLE B 0.01; //Sample second side
9  GB = GROUP SB BY S;
10 GB2 = FOREACH GB GENERATE COUNT_STAR(SB), group;
11 OrderedB = ORDER GB2 BY $0 DESC;
12 PopularB = LIMIT OrderedB k;
13
14 PA = JOIN A BY O LEFT, PopularA BY O USING 'replicated';
15 PPA= JOIN PA BY O LEFT, PopularB BY S USING'replicated';
16 PB = JOIN B BY S LEFT, PopularB BY S USING 'replicated';
17 PPB= JOIN PB BY S LEFT, PopularA BY S USING 'replicated';
18
19 SPLIT PPA INTO APopInA IF PopularA::O is not null,
20    APopInB IF PopularB::S is not null, ANonPop IF
21    PopularA::0 is not null and PopularB::S is not null;
22
23 SPLIT PPB INTO BPopInB IF PopularB::S is not null, BPopInA
24    IF PopularA::O is not null and PopularB::S is null, BNonPop
25    IF PopularA::0 in not null and PopularB::S is not null;
26
27  // Perform replicated joins for popular keys
28  JA= JOIN BPopInB BY S, APopInB BY O USING 'replicated';
29  JB= JOIN APopInA BY S, BPopInA BY S USING 'replicated';
30 // Standard join for non−popular keys
31  JP = JOIN ANonPop BY O, BNonPop BY S;
32
33 $result = UNION ONSCHEMA JA, JB, JP; }
```

that, since there is no communication between nodes after a job execution has started, a skewed key distribution will cause performance problems even if the hit rate for those keys is low. This is because MapReduce will still need to send all the tuples corresponding to these popular keys to a single reduce task.

Our algorithm executes a replicated join for the keys that have a highly skewed distribution in the input and a standard join for the rest. In other words, joining on keys that are responsible for load unbalancing is done by replicating one side on each machine and performing a local hash join. For the remaining keys, the join is executed by grouping the two sides by the join key and assigning the execution of each group to a different machine (as is standard in Pig). In Algorithm 11, we present the Pig Latin code for an example join of expressions A and B*, on positions O and S respectively. Initially, we sample and extract the top-k popular terms for each side (lines 3-12), *PopularA* and *PopularB* respectively. Then, for each side of the input, we perform two left joins to associate tuples with PopularA and PopularB (lines 14-17). This allows us to split each of our inputs to three sets (lines 19-25), marked accordingly in Figure 4.1:

1. The tuples that correspond to keys that are popular on the other side (e.g. for expression A the keys in PopularB). For side B, we put an additional requirement, namely that the key is *not* in PopularB. This is done to avoid producing the results for tuples that are popular twice.

2. The tuples that correspond to keys that are popular on the same side (e.g. for expression A the keys in PopularA).

3. The tuples that do not correspond to any popular keys on either side.

We use the above to perform replicated joins for the tuples corresponding to popular terms and standard joins for the tuples that are not. The tuples in A corresponding to popular keys in A (APopInA) are joined with the tuples in B that correspond to popular keys in A using a replicated join (line 28). The situation is symmetric for B (line 29). The tuples that do not correspond to popular keys from either side are processed using a standard join (line 31). The output of the algorithm is the union of the results of the three joins.

We should note that our algorithm will fail if APopInB and BPopInA are not small enough to be replicated to all nodes. But this can only be true if there are some keys that are popular in both sets. Joins with such keys would anyway lead to an explosion in the result set (since the result size of each of these popular keys is the product of their appearances in each side).

*for brevity, we have omitted some statements that project out columns that are not relevant for our algorithm

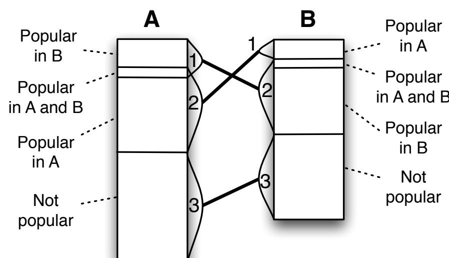


Figure 4.1: Schematic representation of the joins to implement the skew-resistant join

4.2 Evaluation

We present an evaluation of the techniques presented in this chapter using synthetic and real data, and compare our approach to a commercial RDF store.

We have used two different Hadoop clusters in our evaluation: a modest cluster, part of the DAS-4 distributed supercomputer, and a large cluster installed at Yahoo!. The former was used to perform experiments in isolation and consists of 32 dual-core nodes, 4 GB of RAM and 250 GB of local space, on a single disk. The Yahoo! Hadoop cluster that we have used in our experiment consists of over 3500 nodes, each with two quad-core CPUs, 16 GB RAM and 900 GB of local space. This cluster is used in a utility computing fashion and thus we do not have exclusive access, meaning that we can not exploit the full capacity of the cluster and our runtimes at any point might be (negatively) influenced by the jobs of other users. We can thus only report actual, but not best possible performance.

In order to compare our approach with existing solutions, we deployed Virtuoso v7 [23], a top-performing RDF store, on a high-end server: a 4-socket 2.4GHz Xeon 5870 server (40 cores) with 1TB of main memory and 16 magnetic disks in RAID5, running Red Hat Enterprise Linux 6.0.

We chose two datasets for the evaluation. Firstly, the Berlin SPARQL benchmark [14], Business Intelligence use-case v3.1 (BSBM-BI). This benchmark consists of 8 analytical query patterns from the e-commerce domain. The choice for this benchmark is based on the scope of this work, namely complex SPARQL queries from an analytical RDF workload.

Secondly, we also used our engine for some analytical queries on RDF data

Query	1B DAS4	1B Y!	10B Y!
1	38	110	77
2	13	23	31
3	17	18	24
4	38	94	54
5	64	190	112
6	48	34	79
7	26	43	46
8	60	119	98

Table 4.1: Execution time (in minutes) of the BSBM queries on 1B data on the DAS-4 and Yahoo! cluster

that Yahoo! has crawled from the Web. This data is a collection of publicly available information on the Web encoded or translated in RDF. The dataset that we used consists of about 26 billion triples that correspond to about 3.5 terabytes of raw data (328 gigabytes compressed).

4.2.1 Experiments

In our evaluation, we measured:

– *The performance of our approach for large datasets.* To this end, we launched and recorded the execution time of all the queries on BSBM datasets of 1 and 10 billion triples.

– *The effectiveness of our dynamic optimization technique.* To this purpose, we measured the cost of this process and what is its effect on the overall performance.

– *The load-balancing properties of our system.* To this end, we have performed a high-level evaluation of the entire querying process in terms of load balancing, and we have further focused on the performance of the skew-resistant join, which explicitly addresses load-balancing issues.

General performance

We have launched all 8 BSBM queries on 1 billion triples on both clusters and on 10 billion triples using only the Yahoo! cluster. The parameters used to construct them are reported in Appendix B.2.

Query	Cold runtime in sec.	Warm runtime in sec.
1	226 (4m)	108 (2m)
2	41	15
3	1740 (29m)	1445 (24m)
4	3126 (52m)	3055 (50m)
5	702 (11m)	319 (5m)
6	6	0.06
7	50	0.09
8	2398 (40m)	2182 (36m)

Table 4.2: Runtime of the BSBM queries using Virtuoso

In Table 4.1 we report the obtained runtimes. We make the following observations: First of all, the fastest queries (queries 2 and 3) have a runtime of a bit less than 20 minutes on the DAS-4 cluster. The slowest is query 5 that has a runtime of about one hour. For the 1B-triple dataset, the execution times on the Yahoo! cluster are significantly higher than those on the DAS-4 cluster. This is due to the fact that the Yahoo! cluster is shared with other users, so, we often need to wait for the jobs of other users to finish in order to start our execution.

We also note that the runtime does not proportionally increase with the data size on the Yahoo! cluster: the runtimes for the one and ten billion-triple datasets are comparable. Such behavior is explained by the fact that a proportional amount of resources is allocated to the size of the input and the (significant) overhead to start MapReduce jobs does not increase. Our approach, combined with the large infrastructure at Yahoo! allows us to scale to much larger inputs while keeping the runtime fairly constant.

To verify the performance on a real-world scenario and on messy data, we have launched three non-selective SPARQL queries over an RDF web crawl of Yahoo!. The queries are reported in Appendix B.1. The first query is used for identifying "characteristic sets" [57]: frequently co-occurring properties with a subject. The second query identifies all the properties used in the dataset and sorts them according to their frequency. The third query identifies the classes with the most instances. These queries are typical of an exploratory ETL workload on messy data, aimed to create a basic understanding of the structure and interesting properties of a web-crawled dataset.

From the point of view of the computation, the first two queries have non-bound properties and the last one has a very non-selective property (*rdf:type*). Therefore they will touch the entire dataset, including problematic proper-

Query	Cost input extraction	Cost dyn. optimizer	Final query execution
1	358 (6m)	732 (12m)	1183 (19m)
2	262 (4m)	n.a.	545 (9m)
3	346 (6m)	n.a.	718 (12m)
4	382 (6m)	881 (15m)	1015 (17m)
5	457 (7m)	2043 (34m)	1401 (23m)
6	492 (8m)	865 (14m)	1510 (25m)
7	317 (5m)	377 (6m)	843 (14m)
8	487 (8m)	1514 (25m)	1622 (27m)

Table 4.3: Breakdown of query runtimes (in seconds) on 1B BSBM data

ties that cause skew problems. The runtime of these three queries was of respectively 1 hour and 21 minutes, 29 minutes and 25 minutes. The first query required 6 MapReduce jobs to be computed. The second and third each required 3 jobs.

It is interesting to compare the performance for BSBM against a standard RDF store (Virtuoso) even if the approaches are radically different. We loaded 10 billion BSBM triples on the platform described previously. This process took 61 hours (about 2.5 days) and was performed in collaboration with the Virtuoso development team to ensure that the configuration was optimal.

We executed the 8 BSBM queries used in this evaluation and we report the results in Table 4.2. For some queries, Virtuoso is many orders of magnitude faster than our approach (namely, for the simpler queries like queries 1,2, 6 and 7). For the more expensive queries, the difference is less pronounced. However, this performance comes at the price of a loading time of 61 hours, necessary to create the database indexes. To load the data and run all the queries on Virtuoso, the total runtime amounts to 63 hours, while in our MapReduce approach, it amounts to 8 hours and 40 minutes. Although we can not generalize this conclusion to other datasets, the loading time of Virtuoso is not amortized for a single query mix in BSBM-BI.

In this light, the respective advantages of the two systems are in running many cheap queries for Virtuoso and running a limited number of expensive queries for our system. Furthermore, our system can exploit existing Hadoop installations and run concurrently with heterogeneous workloads.

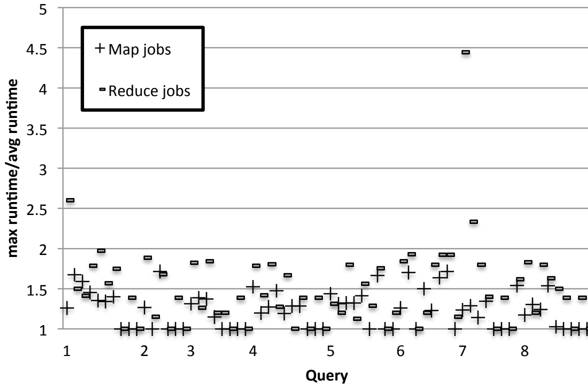


Figure 4.2: Maximum task runtime divided by average task runtime for a query mix

Dynamic optimizer

As discussed in Section 4.1.1, query execution consists of three phases: (a) extraction of triple patterns, (b) identifying the best execution plan using dynamic query optimization and (c) executing the full, optimized, query plan.

In Table 4.3, we summarize the execution time of each of these three phases for the 8 BSBM queries on the DAS-4 cluster, using the one billion triples dataset. we observe that extracting the input patterns is an operation that takes between 4 and 8 minutes, depending on the size of the patterns. Furthermore, the cost of dynamic optimization is significant compared to the total query execution time and largely depends on the complexity of the query, although, in our approach, part of the results are calculated during the optimization phase.

Load balancing

As the number of processing nodes increases, load-balancing becomes increasingly important. In this section, we present the results concerning load balancing properties for our approach.

Due to the synchronization model of MapReduce, it is highly desirable that there are no tasks that take significantly longer than others. In Figure 4.2, we present the maximum task execution time, divided by the average task execution time for all the jobs launched to process a query mix on the DAS-4

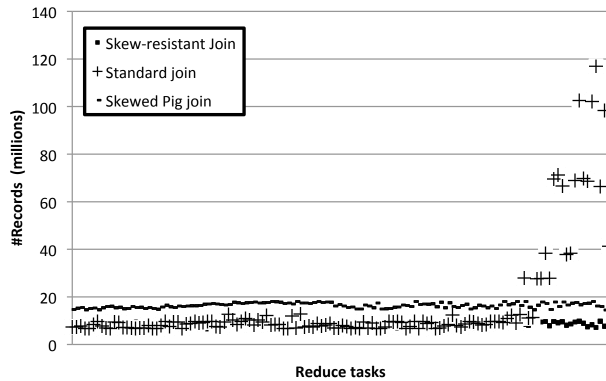


Figure 4.3: Comparison of load distribution between the skew-resistant join and the standard Pig join

cluster. The x axis corresponds both to time and the queries that correspond to each job, the y axis corresponds to the time it took to execute the slowest task divided by the time it took to execute a task, on average. In a perfectly load-balanced system, the values in y would be equal to 1. In our system, besides a single outlier, the slowest tasks generally take less than twice the average time, indicating that the load balancing of our system is good. A second observation is that the load imbalance is higher in the reduce jobs. This is to be expected, considering that, in our system, the map tasks are typically concerned with partitioning the data (and process data chunks of equal size), while the reduce tasks process individual partitions.

Data skew becomes increasingly problematic as the number of processing nodes increases, since it generates unbalance between the workload of each node [45]. In the set of experiments described in this section, we analyze the performance of the skew-resistant join that we have introduced in Section 4.1.3 to efficiently execute joins on data with high skew.

To this purpose, we launched an expensive join using the 1 billion BSBM dataset and we analyzed the performance of the standard and the skew-resistant join. Our experiments were performed on the DAS-4 cluster, since we required a dedicated cluster to perform a comparative analysis. Considering that this platform uses only 32 nodes in total, the effect on the Yahoo! cluster would have been much more pronounced (since it is several orders of

magnitude bigger).

We launched a join that used the predicate of the triples as a key; namely, we have performed a join of pattern *(?s ?p ?o)* with pattern *(?p ?p1?o1)*. Such joins are common in graph analysis, dataset characterization and reasoning workloads (e.g. RDFS rules 2-3 and 7).

The runtime using the classical join was of about 1 hour and 29 minutes. On the other side, the runtime using the skew-resistant join was of about 57 minutes. Therefore, such a join has a consistent impact on the performance, in case there is significant skew in the data. The impact is even higher if we consider that the skew-resistant join requires 21 jobs to finish while the classical job can be encoded using a single MapReduce job.

The reason behind such increase of performance lies in the way the join is performed. With the skew-resistant join, all the joins between keys that are popular in the two sides are performed using a replicated join, which is efficient because it does not involve communication between the nodes. On the other hand, if we use the classical method provided by Pig all the computation is performed in the reduce phase and this is deleterious if there is a large difference between the cardinalities of the join. In Figure 4.3 we report the number of records received in input in the reduce phase with both methods. We see that some reducers receive a much larger number of tuples than others (these are the ones at the end of the x axis). This implies that some nodes will need to perform much more computation than others. With the skew-resistant join, all the joins among popular terms are performed in the map phase and as result all the reducers receive a similar number of tuples in the input.

We also report reduce task runtimes for the skew-resistant join implementation in Pig, which calculate a histogram for join keys in order to better distribute them across the join tasks. Although the size of the cluster is small enough to ensure an even load-balance using this method, we note that the standard Pig skewed join sends almost double the number of records to each reduce task. This is attributed to fact that our approach shifts much of the load for joining to the Map phase.

4.3 Related Work

We have compared our approach with previous results from three related areas: (i) MapReduce query processing (ii) adaptive and sampling-based query optimization and (iii) cluster-aware SPARQL systems.

From the debate on the relative efficiency of parallel relational DBMS query processing versus MapReduce [75] we know that, if time and resources are

available to import and analyze the data, the former kind of systems can reach higher efficiency, given the data is structured and the query is regular. Investing into partitioning, index building and statistics gathering makes perfect sense in those scenarios where data once imported is re-used many times. However, there are use cases, such as web-scale data analysis and ETL, where data preparation time dominates, and there may not be room for amortization. Our work is intended for situations where the ability of MapReduce clusters to process huge volumes of raw data with small preprocessing is ideal.

We base our system on the Pig framework [63], which is built on top of Hadoop, that is a popular open-source MapReduce implementation. Pig is a high-level programming environment for MapReduce, comparable in abstraction level to relational algebra, and thus is a suitable target language for the generation of query plans. Pig comes with a rich library of operators, some of which are directed at handling situations with data skew [29].

In the relational context, similar efforts towards SQL query processing over a MapReduce cluster are e.g. Hive [80] and HadoopDB [4]. Both projects do not provide query optimization when data is raw and unprocessed. Data import is a necessary first step in HadoopDB and may be costly. In Hive, query optimization based on statistics is only available if the data has been analyzed as a prior step.

An interesting approach to on-the-fly query optimization in MapReduce is Manimal [40] which analyzes MapReduce jobs on-the-fly and tries to enhance them by inserting compression code and sometimes even on-the-fly indexing and index lookup steps.

Situations where there is absence of data statistics in the relational context of query optimization has led to work on sampling and run-time methods. Our work reuses the Bi-Focal sampling algorithm [28] which came out of the work in the relational community to use sampling for query result size estimation. In this work, we have adapted the bi-focal algorithm using the Pig language.

The fuzzy structure in semi-structured data models (such as XML and RDF) often leads to situations where there are no good statistics. Here, we often find that certain regularities, which in the relational context are explicit in the schema, re-surface as data correlations, which are hard to capture in statistics. We note that the recent work on Characteristic Sets [57] provides an interesting path forward for the case of RDF. To provide robust query optimization in the face of data correlations and absence of statistics that capture them, ROX [2, 41] proposes to use run-time sampled query evaluation that dynamically observes intermediate result sizes and optimizes the query plan during execution. Our work brings some of these ideas to the MapReduce context. The rigid structure of MapReduce and high latencies in starting new jobs, led

us to adjust the dynamic re-optimization strategies to these constraints. Other interesting run-time approaches are sideways-information passing [58] in large RDF joins. These are not easily adaptable to the constraints of MapReduce.

With the ever growing sizes of RDF data available, scalability has been a primary concern and major RDF systems such as Virtuoso [23], 4store [68], and BigData [10] have evolved to parallel database architectures targeting cluster hardware. RDF systems typically employ heavy indexing, going as far as creating replicated storage in all six permutations of triple order [59, 96], which makes data import a heavy process. Such choice puts them in a disadvantage when the scenario involves processing on huge amounts of raw data. As an alternative to the parallel database approach, there are several other projects that process SPARQL queries over MapReduce. PIGSparQL [69] performs a direct mapping of SPARQL to Pig without focusing on optimization. RAPID+ [67] former provides a limited form of on-the-fly optimization where *look-ahead processing* tries to combine multiple subsequent join steps. The adaptiveness of this approach is however limited compared to our sampling based run-time query optimization.

4.4 Conclusions

The RDF data model is gaining popularity on the Web and an increasing amount of data is being released in this format. Large web companies like Yahoo! are actively crawling RDF data from the Web and need to efficiently process and query it. There are several challenges in performing scalable RDF data processing. In this chapter, we address some of them, and to this purpose we have presented an engine for the processing of complex analytic SPARQL 1.1 queries, based on Apache Pig.

More in particular, we have developed: (i) a translation from SPARQL 1.1 to Pig Latin, (ii) a method for runtime join optimization based on a cost model suitable for MapReduce-based systems, (iii) a method for result set estimation and join key skew detection, and (iv) a method for skew-resistant joins written in Pig. We have evaluated our approach on a standard and a very large Hadoop cluster used at Yahoo! using synthetic benchmark data and real-world data crawled from the Web.

In our evaluation, we established that our approach can answer complex analytical queries over very large synthetic data (10 Billion triples from BSBM) and over the largest real-world messy dataset in the literature (26 Billion triples). We compared our performance against a state-of-the-art relational database-backed RDF store on a large-memory server, even though the two

approaches bear significant differences. While our approach is not competitive in terms of query response time, our system has the advantage that it does not require a-priori loading of the data, and thus has far better loading plus querying performance. Furthermore, our system runs on a shared architecture of thousands of machines, significantly easing deployment and potentially scaling to even larger volumes of data.

We verified that the load in our system is well-balanced and our skew-resistant join significantly outperforms the standard join of Pig for skewed key distributions in the input.

In this work, for the first time, it has been shown that MapReduce is suited for very large-scale analytical processing of RDF graphs and it is, in fact, better suited than a traditional RDF store for a setting where a relatively small number of queries that require a high computation will be posted on a very large dataset. These queries are typically used in a ETL workload, where often the user needs to launch expensive queries to extract and transform very large volumes of data. The computation required by such queries has motivated our choice of using MapReduce, and by doing that we also gain the additional advantage that we avoid an expensive preprocessing phase, which is not necessary in a typical ETL scenario.

We see future work in optimizing our architecture to further reduce the overhead of the framework. This can be achieved by replacing sequences of standard Pig operators with specialized operators. Moreover, we can turn to an approach that adaptively indexes part of the input, so as to reduce the number of jobs. In many cases, some of the generated MapReduce jobs were very short and they can be implemented more efficiently without Pig.

Although in this chapter we have presented the algorithm to handle skewed joins in the context of Pig, we expect that the result is transferable to a general parallel data-processing framework. Furthermore, we note that parts of the skew-resistant join can be already calculated during Bi-focal sampling (e.g. sampling and extracting the popular terms for the input relations). Both issues merit additional investigation.

Given the significant cost of dynamic optimization in Pig, another direction for future work is to research techniques to predict the query behavior to limit or avoid the sampling and pruning phases that trigger the execution of additional MapReduce jobs.

Summarizing, in this chapter, we have presented a technique with which technologies like MapReduce and Pig can be efficiently employed for large-scale SPARQL querying. The presented results are promising and set the lead for a new viable alternative to traditional RDF stores for executing expensive analytical queries on large volumes of RDF data.

Part II

Reasoning at query time

Chapter 5

Hybrid-reasoning

In the previous part we have described a technique to perform forward-chaining reasoning using the MapReduce programming model. By precomputing all the inference, such technique enables efficient responses at query time, but at the cost of an expensive up front closure computation, which needs to be redone at every update of the knowledge base.

Backward-chaining does not need such an expensive and change-sensitive precomputation, and is therefore suitable for more frequently changing knowledge bases, but has to perform more computation at query time. In this chapter we address this last type of reasoning and we present a general hybrid algorithm to perform backward-chaining reasoning on very large RDF datasets. Our method materializes a fixed set of selected queries, *before* query time, whilst applying backward chaining *during* query time. This *hybrid* approach is a trade off between a reduction in rule applications at query time and a small, query independent computation of data before query time. Our method relies on a backward-chaining algorithm to calculate the inference which exploits such partial materialization and the parallel computing power of modern architectures.

In this chapter, we tackle two problems: First we focus on the problem of determining whether the general hybrid reasoning algorithm we propose is correct, i.e. it terminates, is sound and complete. We shall argue that the

correctness is not dependent on a particular rule set, but holds for a generic rule set that can be expressed in Datalog.

For the evaluation, however, we will apply our method to the OWL RL rule set, which is the latest standard OWL profile designed to work on a large scale. We address some crucial challenges that arise with OWL RL and propose a set of novel optimizations that substantially improve the computation and hence the execution time.

We have implemented these techniques in an experimental prototype called QueryPIE, and we have tested the performance using artificial and realistic datasets of a size between five and ten billion triples. The evaluation shows that we are able to perform OWL reasoning using one machine equipped with commodity hardware which keeps the response time often below one second.

The remainder of this chapter is organized as follows: in Section 5.1 we present at high level the main idea behind hybrid reasoning. The purpose of this section is to introduce the reader to our problem and to provide a high level overview of our approach.

In Section 5.2 we will describe the backward-chaining algorithm that is used within our method to calculate the inference. Section 5.3 formalizes the precomputation algorithm of hybrid reasoning and proves its correctness. Next, in Section 5.4, we focus on the execution of the OWL 2 RL/RDF rule set (for simplicity we will refer to it as the OWL RL rule set) presenting a series of optimizations to improve the performance on a large input. In Section 5.5 we present an evaluation of our approach using single pattern queries on both realistic and artificial data. In Section 5.6 we report on related work. Finally, Section 5.7 concludes and gives some directions for future work.

5.1 Hybrid reasoning: Overview

In principle there are two different approaches to infer answers in a database with a given ruleset: One is to compute the complete extension of a database under some given ruleset *before* query time (the method presented in Chapter 2 is an example of it) and the other is to infer only the necessary entries needed to yield a complete answer from the ruleset on-demand, i.e. *at* query time.

The former's advantage is that querying reduces, after the full materialization, to a mere lookup in the database and is therefore very fast compared to the latter approach, where for each answer a proof-tree has to be built.

On the other hand, if the underlying database changes frequently, ex-ante materialization has a severe disadvantage as the whole extension must be recomputed with each update. In this case, an on-demand approach has a clear

advantage.

The approach presented in this chapter positions itself in between: the answers for a carefully chosen set of queries are materialized before query time and added to the database. Answers to queries later posed by the user are inferred at query time.

Traditionally, each approach has been associated with an algorithmic method to retrieve the results: Backward chaining was specifically aimed at on-demand retrieval of answers, only materializing as little information as necessary to yield a complete set of answers, whilst forward chaining applies the rules of the given ruleset until the closure is reached.

Since we want to avoid complete materialization of the database, and therefore are only interested in specific answers, we use backward chaining in both cases: we use backward chaining to materialize only the necessary information for the carefully chosen queries which we then add to the database, and we use backward chaining to answer the user queries.

To this end, we introduce a backward-chaining algorithm which exploits parallel computing power and the fact that some triple patterns are pre-materialized to improve the performance. For example, if one of these pre-materialized queries is requested at query-time, the backward-chaining algorithm does not need to build the proof-tree, but a lookup suffices. In case the pre-materialized patterns frequently appear at user query-time, such optimization is particularly effective.

To give an idea on how this works, consider the following:

Example 1. *Consider the two following rules from the OWL RL ruleset:*

$$\begin{array}{ll} T(a,p1,b) & \leftarrow T(p,SPO,p1) \wedge T(a,p,b) \\ T(x,SPO,y) & \leftarrow T(x,SPO,w) \wedge T(w,SPO,y) \end{array}$$

where we use from now on the abbreviation SPO for reasons of space and a, b, p, $p1$, x, y, w are variables (a list of all the abbreviations used in this chapter is contained in Table 5.1).

*Assume we want to suppress the unfolding of all atoms of the form $T(x,SPO,y)$, modulo variable renaming. If we use Datalog to implement these rules in a program, then we can replace each atom by some new atom, using an extensional database predicate (*edb*)*, say S. After the substitution, Example 1 would become:*

$$\begin{array}{ll} T(a,p1,b) & \leftarrow S(p,SPO,p1) \wedge T(a,p,b) \\ T(x,SPO,y) & \leftarrow S(x,SPO,w) \wedge S(w,SPO,y) \end{array}$$

*In Datalog, edb predicates are predicates that do not appear in the head of any rule. Therefore, they can be implemented much more efficiently since they require only a single lookup in the database.

Abbreviation	Full text
TYPE	*rdf:type*
SCO	*rdsf:subClassOf*
SPO	*rdsf:subPropertyOf*
EQC	*owl:equivalentClass*
EQP	*owl:equivalentProperty*
INV	*owl:inverseOf*
SYM	*owl:SimmetricProperty*
TRANS	*owl:TransitiveProperty*
INTER	*owl:intersectionOf*

Table 5.1: List of abbreviations for common URIs used in this chapter.

Clearly, the two programs in Example 1 do not yield the same answers for T anymore. To restore this equality we need to calculate all "$T(x, SPO, y)$"-triples and add them to the auxiliary relation named S in the database. In our example this would mean that S contains the transitive closure of all "$T(x, SPO, y)$"-triples which are inferable under the ruleset in the database.

Notice that "$T(x, SPO, y)$"-triples can also be derived with the first rule if $p1 = SPO$. Furthermore, if S indeed contains the transitive closure of all "$T(x, SPO, y)$"-triples the second rule can be rewritten as $T(x, SPO, y) \leftarrow S(x, SPO, y)$.

Before we formalize this method and show that it is indeed harmless in the sense that everything which could be inferred under the original program can be inferred under the altered program and vice versa, we shall discuss the backward-chaining algorithm we use and how it exploits the prematerialization for an efficient execution. After this, we will formally discuss the correctness of our method.

5.2 Hybrid Reasoning: Backward-chaining

In the current and following sections, we will use the notation and notions that come from the Datalog theory, and more in particular from [3, Chapter 12], to formalize and to prove the correctness of our method.

To ease the understanding of our explanation, we will briefly recall some well-known notions of Datalog that will be frequently used. To the same purpose, we will also enrich our explanation with brief examples in order to

facilitate the comprehension in case the reader is not completely familiar with the concepts that are being used.

Let $\mathfrak{I}$ be a generic Datalog database and R be a predicate symbol of arity n. We denote with $R^{\mathfrak{I}}$ the n-ary relation named R in $\mathfrak{I}$. In a similar fashion we denote with $q(\bar{x})^{\mathfrak{I}}$ the set of all answers to a Datalog query $q(\bar{x})$ in $\mathfrak{I}$.

In our formalization, we denote T_P as the *immediate consequence operator* of the Datalog program P. An immediate consequence operator is an operator that maps a database $\mathfrak{I}$ to the database $T_P(\mathfrak{I})$, where $T_P(\mathfrak{I})$ is $\mathfrak{I}$ extended by all facts that could be inferred from facts in $\mathfrak{I}$ under P. We define $T_P^0(\mathfrak{I}) := \mathfrak{I}$ and $T_P^{n+1} = T_P \circ T_P^n$.

With ω we indicate the first infinite limit ordinal and set $T_P^{\omega}(\mathfrak{I}) := \bigcup_{n<\omega} T_P^n(\mathfrak{I})$. According to [3, Chapter 12], we have $P(\mathfrak{I}) = T_P^{\omega}(\mathfrak{I})$ and in particular that for every fact $\bar{a} \in R^{P(\mathfrak{I})}$ there must be some $n < \omega$ such that $\bar{a} \in R^{T_P^n(\mathfrak{I})}$.

We will now discuss our backward-chaining algorithm. The purpose of the backward-chaining algorithm is to derive all possible triples that are part of a given input query Q, given a database D and a ruleset R.

Traditionally, users interact with RDF datasets using the SPARQL language [66] where all the triple patterns that constitute the body of the query are joined together according to some specific criteria. For the moment, we do not consider the problem of efficiently joining the RDF data and focus instead on the process of retrieving the triples that are needed for the query. Therefore, we target our reasoning procedure at *atomic* queries, e.g.,

$$(?\mathtt{c}_1\ \mathtt{rdfs{:}subclassOf}\ ?\mathtt{c}_1)$$

where question marks indicate variables.

Generic inference rules like the ones in the OWL RL ruleset can trivially be rendered into a positive Datalog program as already witnessed in Example 1. The algorithm that we present is inspired by the well-known algorithm QSQ (Query-subquery) that was first introduced in 1986 which generalizes the SLD-resolution technique [88] by applying it to sets of tuples instead of single ones. The variations that we introduce are meant to exploit the computational parallelism that is possible to obtain by using modern architectures.

The QSQ algorithm recursively rewrites the initial query into many subqueries until no more rewritings can be performed and the subqueries can only be evaluated against the knowledge base.

Example 2. *For example, suppose that our initial query is*

$$T(x, rdf:type, Person)$$

and that we have a generic database D and the OWL RL ruleset as R. Initially, the algorithm will determine which rules can produce a derivation

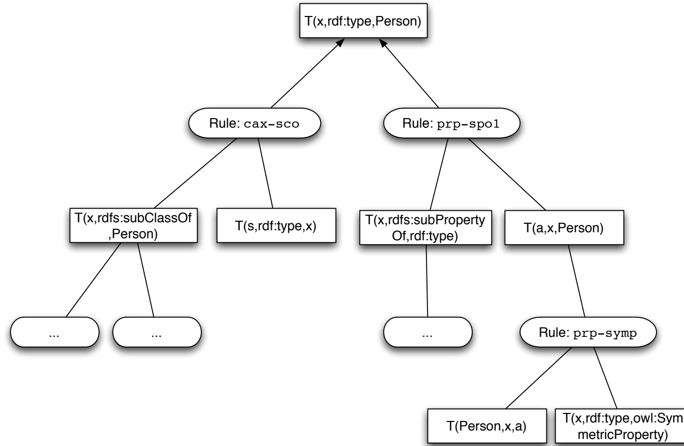


Figure 5.1: Example of a proof tree using the OWL RL rules and with the input query $T(x, rdf : type, Person)$

that is part of the input query. For example, it could apply the subclass and subproperties inheritance rules (`cax-sco` and `prp-spo1` in the OWL RL ruleset). After it has determined them, it will move to the body of the rules and proceed evaluating them. In case these subqueries will produce some results, the algorithm will execute the rules and return the answers to the upper level.

With this process, we create a tree that has the original query as root and the rules and subqueries that might contribute to derive some answers as the internal nodes. This tree is normally referred as proof tree *because it represents all the derivation steps that are taken to derive answers of our initial query starting from some existing facts. In Figure 5.1 we report an example of such a tree for our example query.*

An important problem of backward-chaining algorithms concerns the execution of recursive rules. Recursive rules and more in general cycles in the proof tree are an important threat since they could create loops in the computation that the algorithm must handle.

The QSQ algorithm guarantees termination even in presence of recursive rules by memorizing in a global data structure all the subqueries already evaluated and avoiding to make a recursive call with a query if this was already previously done. This means that eventual derivations that require the evaluation of the same query more times cannot be inferred because the algorithm

will stop the recursion after one application.

To solve this issue, the algorithm repeats the execution of the query until fix closure (all derivations are produced). This operation is performed at every recursive call, to ensure that all the bindings for each subquery in the proof tree are correctly retrieved.

It has been proved that the QSQ algorithm is sound and complete [89]. Because of this, we are ensured that with this methodology no derivation will be missed.

The original version of this algorithm is hard to parallelize because it requires a sequential execution to build the proof tree with a depth-first strategy and it exploits the access to a global data structure to remember all the previously derived queries. Therefore, its execution is unable to take advantage of modern multi-core architecture or clusters of several independent nodes.

Because of this, in the next section we will present an adaptation of this algorithm to our specific use case so that it can be more easily parallelized and show that the fundamental properties of termination and completeness are still valid. After this, in Section 5.2.2, we will describe how we can exploit the precomputation of some queries to increase the performance of backward-chaining.

5.2.1 Our approach

We introduced two key differences to improve the parallelization of the computation:

- Instead of constructing the proof tree sequentially using a depth-first strategy as the original QSQ algorithm does, we do it in parallel by applying the rules on separate threads and in an asynchronous manner. For example, if we look back at Figure 5.1, the execution of rules `cax-sco` and `prp-spo1` is performed concurrently by different threads. This execution strategy makes the implementation and the maintenance of the global data structure, used for the caching of previous queries, difficult and inefficient.We hence choose to replace this mechanism to only remember which queries where already executed on the single paths of the tree. While such a choice might lead to some duplicate answers because the same queries can be repeated more times, it allows the computation to be performed in parallel limiting the usage of expensive synchronization mechanisms;
- Because the proof tree is built in parallel, ensuring completeness by having a loop at every recursive call is inefficient since the same query can

appear multiple times on different parts of the tree. Therefore, we replace it with a global loop that is performed only at the root level of the tree and storing at every iteration all the intermediate derivations.

We report the algorithm using pseudocode in Algorithm 12. In the pseudocode, we use the relation $\sqsubseteq$ to define whether one query is more specific than another. In this case, all the results of the more specific query are contained in the answer set of the most generic one.

More formally, we can define it as follows:

Let $\bar{t} := (t_1, \ldots t_n)$ and $\bar{t}' := (t'_1, \ldots t'_n)$ be tuples with $t_i, t'_i \in \mathsf{TERM}$, i.e. each component is either a variable or a constant. Then $\bar{t}$ is an instance of $\bar{t}'$, $\bar{t} \sqsubseteq \bar{t}'$, if there is a substitution $\sigma : TERM \longrightarrow TERM$ such that $\sigma(c) = c$ for each constant c and $(\sigma(t'_1), \ldots \sigma(t'_n)) = (t_1, \ldots, t_n)$.
Additionally, if $R(\bar{t})$ and $R(\bar{t}')$ are atoms we define $R(\bar{t}) \sqsubseteq R(\bar{t}')$ iff $\bar{t} \sqsubseteq \bar{t}'$.

Example 3. $(x, TYPE, SYM) \sqsubseteq (x, TYPE, y)$ *where* x, y *are variables and* $TYPE$ *and* SYM *are the abbreviation in Table 5.1.*
Also $(x, TYPE, y) \sqsubseteq (y, TYPE, x)$.

If $R(\bar{t}) \sqsubseteq R'(\bar{t}')$ and $R'(\bar{t}') \sqsubseteq R(\bar{t})$ then $R(\bar{t})$ equals $R'(\bar{t}')$ up to variable renaming.

The procedure *main* is the main function used to invoke the backward-chaining procedure for a given atomic query Q. It returns the derived answers for the input query. The procedure consists of a loop in which the recursive function *infer* is invoked with the input query. This function returns all the derived answers for Q that were calculated by applying the rules using backward-chaining (line 5) and all the intermediate answers that were inferred in the process, saved in the global variable *Tmp*. In each loop-pass the latest results in *Tmp* and *New* are checked against the accumulated answers of the previous runs in *Mat* and *Database*. If nothing new could be derived the loop terminates.

After this loop has terminated, the algorithm returns *New* (line 7) which contains after the last loop-pass all answers to the input query (cf. line 13) and the results.

The function *infer* is the core of the backward-chaining algorithm. Using the function *lookup*, it first retrieves for the formal parameter Q all answers which are facts in the database or were previously derived (line 13). After this, it determines the rules that can be applied to derive new answers for Q (lines 14–15) and calculates the substitution θ to unify the head of the applicable rule with the query Q (line 16). It proceeds with evaluating the body of the rule

Algorithm 12 Backward-chaining algorithm:
Database and *RuleSet* are global constants, where *Database* is a finite set of facts and *RuleSet* is a Datalog program. *Tmp* and *Mat* are global variables, where *Mat* stores results of the previous materialization round and *Tmp* stores the results of the current round. The parameter Q represents an input pattern. Both functions *main* and *infer* return a set of triples, whilst the function *lookup* returns a set of substitutions. θ_ε is the empty substitution. Notice that we say $Q \in PrevQueries$ iff there is $Q' \in PrevQueries$ s.t. $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$

```
1   function main(Q)
2     New, Tmp, Mat := ∅
3     repeat
4       Mat := Mat ∪ New ∪ Tmp
5       New := infer(Q, ∅)
6     until New ∪ Tmp ⊆ Mat ∪ Database
7     return New
8   end function
9
10  function infer(Q, PrevQueries)
11
12    //This cycle is executed in parallel
13    all_subst := lookup(Q,Database ∪ Mat)
14    for (∀ r ∈ RuleSet s.t. Q is unifiable
15           with r.HEAD and Q ∉ PrevQueries)
16      θ_h := MGU(Q,r.HEAD)
17      subst := {θ_ε}
18      for ∀ p ∈ r.BODY
19        tuples := infer(θ_h(p),PrevQueries ∪ Q)
20        Tmp := Tmp ∪ tuples
21        subst := subst ⋈ lookup(θ_h(p),tuples)
22      end for
23      all_subst := all_subst ∪ subst
24    end for
25
26    return ⋃_{θ∈all_subst} θ(Q)
27
28  end function
```

(lines 16–23) storing in *tuples* and *Tmp* the retrieved answers (lines 19–20), and performing the joins necessary according to the rule body (line 21).

The algorithm copies all the substitutions that were derived by applying the rules into the variable *all_subst* and constructs a set of answers using these substitutions θ (line 26). This set is then returned to the function caller. After the whole recursion tree has been explored exhaustively, *infer* returns control to the function *main*, where the derived answers are copied into the variable *New*. The process is repeated until the closure is reached.

To help the understanding of this algorithm and more in particular of the function *infer*, consider the following example:

Example 4. *Suppose that we have a program that consists of a single rule:*

$$r := T(a, p1, b) \quad \leftarrow T(p, SPO, p1) \wedge T(a, p, b)$$

and the input query is $Q := (x, TYPE, y)$ *where* x *and* y *are variables.*

At the beginning, the function `main` *will pass the query* Q *to the function* `infer`*, which will look for all the rules with an head that is unifiable with* Q *(lines 14–15). In our example, only rule* r *satisfies this condition, and the program calculates the MGU* θ_h *(line 16). An example of such MGU could be* $\theta_h := \{a/x, p1/TYPE, b/y\}$

Then, for each literal in the body of the rule, the algorithm unifies the atoms in the body with the initial query using the MGU θ_h *and retrieves the triples invoking the function* `infer`*. In our example, the first query would be* $(p, SPO, TYPE)$*, and the results would be stored in the variable* `subst`*. Then, the algorithm will launch the execute the second query* (a, p, b) *(notice however that in this case the implementation is aware of all the possible* p*). After this, the results will be joined with the previous ones (line 21), and the triples will be returned to the function* `main`*, which will repeat the execution until all triples are calculated.*

We will now discuss the correctness of our algorithm, which are termination, soundness and completeness.

Termination.

It is easy to verify that the backward-chaining algorithm in Algorithm 12 always terminates. The only two sources for not-termination are (i) the loop in lines 3–6 and (ii) the recursive call in line 19. The first loop will continue until neither New nor Tmp will contain new answers. This happens latest when every relation is equal to the cartesian product of its arity over the domain of the database, hence within finitely many steps.

The recursive call in line 19 will be fired only if there is no Q' in the set *PrevQueries* (see condition in line 15) such that $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$ (cf. page 96), meaning that Q equals Q' up to variable renaming. However, for similar reasons as before there are, up to variable renaming, only finitely many different atomic queries over the domain of the database and so the recursion will terminate.

Soundness.

The soundness is immediate, as new facts can only be derived through rule-application in the for-loop beginning in line 14. Hence, if $R(a_1,\ldots,a_n)$ is a fact derived by the function $infer$, $R(a_1,\ldots,a_n)$ is a fact in $P(\mathfrak{I})$, the least fixpoint model for the Datalog program P and the database $\mathfrak{I}$. Hence Algorithm 12 is sound.

Completeness.

In order to show the completeness of Algorithm 12, we prove Proposition 2, which holds in particular for all answers to Q derived under a given ruleset *RuleSet* and a database *Database*. We first show

Proposition 1. *Let Q be a query for function* `main` *and $R(a_1,\ldots,a_n)$ a fact, which appears in the proof-tree of some fact derived from Q under RuleSet in Database. Then there is some subsequent non-blocked query Q_n appearing in the computation of $infer(Q,\emptyset)$, such that $R(a_1,\ldots,a_n)$ is an answer to Q_n derived under RuleSet in Database.*

Proof. We have to show, that there is a sequence of queries $Q_0,\ldots,Q_n$ such that

1. $Q = Q_0$ and $R(a_1,\ldots,a_n)$ unifies with Q_n
2. for each $i \in \{0,\ldots,n\}$ there is a rule such that Q_i unifies with the head of some rule $r \in ruleSet$ and Q_{i+1} unifies with some body-atom of r,
3. no query is blocked, i.e. there is no subsequence $Q_i \ldots Q_k$ with $0 \leq i < k \leq n$ such that Q_i is up to variable renaming equal to Q_k ($Q_i \sqsubseteq Q_k$ and $Q_k \sqsubseteq Q_i$).

In this case $infer(Q, Ruleset, \emptyset)$ will eventually produce the query Q_n (cf. lines 14-24, of Algorithm 12).

Let $Q(b_1,\ldots,b_m)$ be the fact which unifies with the input query Q in which proof-tree $R(a_1,\ldots,a_n)$ appears. Then there is a sequence of rule applications $r_0,\ldots,r_n$ such that $Q(b_1,\ldots,b_m)$ unifies with the head of r_0, for all $i \in \{1,\ldots,n\}$ some body-atom b_{i,k_i} of r_i unifies with the head of r_{i+1} and $R(a_1,\ldots,a_n)$ unifies with some body-atom $b_{n,\ell}$ of r_n.

Since $Q(b_1,\ldots,b_m)$ was an answer to Q, they unify and so Q unifies with the head of r_0 yielding $\theta_0 := MGU(Q,r_0)$. For all $i \in \{1,\ldots,n\}$ the body-atom b_{i,k_i} of r_i unifies with the head h_{i+1} of r_{i+1} yielding $\theta_{i+1} :=$

$MGU(Q_i, h_{i+1})$ where $Q_i := \theta_i(b_{i,k_i})$ so that we finally reach the body atom $b_{n,\ell}$ of r_n where $Q_n = \theta_n(b_{n,\ell})$ is the query which unifies with $R(a_1, \ldots, a_n)$.

We hence obtain a sequence $Q_0 \ldots Q_n$ satisfying items 1 and 2. We shall show that for every sequence satisfying items 1 and 2 there is a sequence $Q'_0, \ldots, Q'_m$ satisfying items 1–3:

The claim is clear, if the sequence is of length 1, i.e. $n = 0$: Q_0 is never blocked. Let $Q_0 \ldots Q_n$ be a sequence of length $n+1$ with Q_i equals Q_k up to variable renaming where $0 \leq i < k \leq n$. Then the head of r_{k+1} unifies with the query Q_i. The sequence $Q_0, \ldots, Q_i, Q_{k+1} \ldots Q_n$ is properly shorter than $Q_0 \ldots Q_n$ and satisfies items 1–2. The induction hypothesis yields a sequence $Q'_0, \ldots, Q'_m$ which satisfies items 1–3. □

Proposition 2. *Let Q be an input query for function* `main` *and $R(a_1, \ldots, a_k)$ a fact, which appears in the proof-tree of some fact derived from Q under RuleSet in Database. Then there is a repeat-loop pass from which onwards $R(a_1, \ldots, a_k)$ is returned by every query Q_n which unifies with $R(a_1, \ldots, a_k)$.*

Proof. We prove by induction upon $n < \omega$, that the fact $R(a_1, \ldots, a_k)$ is yielded in at most n repeat-loop passes, if the height of the minimal proof tree for $R(a_1, \ldots, a_k)$ is equal to n.

Proposition 1 shows that $infer(Q, \emptyset)$ produces an unblocked query Q_n such that $R(a_1, \ldots, a_k)$ unifies with Q_n.

If the proof tree is of height 0, then $R(a_1, \ldots, a_k)$ is a fact in $Database$ and $infer$ will always produce this fact in the look-up of line 13 which will be returned (cf. line 26) by $infer$ for all repeat-loop passes.

Assume the proof tree is of height > 0. Then there is a rule $r : R(\bar{t}) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m)$ and a variable assignment β such that $R(\beta(\bar{t})) = R(a_1, \ldots a_k)$ and for each $i \in \{1, \ldots, m\}$ the fact $R_i(\beta(\bar{t}_i))$ has a proof tree of height at most n under $RuleSet$ in $Database$.

The head h of r unifies with $R(a_1, \ldots, a_k)$. Let $\theta := MGU(Q_n, h)$ then each $R_i(\beta(\bar{t}_i))$ unifies with $Q'_i := R_i(\theta(\bar{t}_i))$. Since Q_n is not blocked, every Q'_i is a subsequent query of Q_n.

By the induction hypothesis, for all $i \in \{1, \ldots, n\}$, every Q'_i occurring in the computation yields $R_i(\beta(\bar{t}_i))$ after at most n repeat-loop passes. Hence $R(a_1, \ldots, a_n)$ is returned by this particular Q_n at the very latest in the n-th repeat-loop pass and eventually added to Mat (cf. line 4) so that every subsequent query that unifies with $R(a_1, \ldots, a_n)$ will return this fact as look-up in line 13.

□

By proving Proposition 2, we have shown that Algorithm 12 is complete in a way that given a generic ruleset and database, the algorithm is able to derive all possible conclusions that can be derived. However, in our approach this algorithm is invoked with a ruleset that is different from the one that should be used in first place. Therefore, we still need to prove that our entire approach is complete in a sense that an execution of this algorithm with the modified rule set retrieves the same results than an execution that uses the original ruleset (provided that a prematerialization is performed beforehand). This issue will be discussed in Section 5.3.

In the following section, we will conclude our discussion of our backward-chaining algorithm by describing how a generic set of the precalculated predicates can be used in the implementation to speed up the performance of reasoning at query time.

5.2.2 Exploiting the precomputation for efficient execution.

In the previous section, we made no difference in the description of our backward-chaining algorithm between subqueries that are precomputed or not. However, the pre-materialization of a selection of queries allows us to substantially improve the implementation and performance of backward-chaining performance by exploiting the fact that these queries can be retrieved with a single lookup.

In our implementation, these queries are maintained in memory so that the joins required by the rules can be efficiently executed. Also, the availability of the pre-materialized queries in memory allows us to implement a very efficient information passing strategy to reduce the size of the proof tree by identifying beforehand whether a rule can contribute to derive facts for a given query.

In fact, the pre-materialization can be used to determine early failures: Emptyness for queries which are subsumed by the pre-materialized queries can be cheaply derived since a lookup suffices. Therefore, when scheduling the derivation of rule body atoms, we give priority to those body atoms that potentially match these pre-materialized queries so that if these "cheap" body atoms do not yield any facts, the rule will not apply, and we can avoid the computation of the more expensive body atoms of the rule for which further reasoning would have been required.

To better illustrate this concept, we proceed with an example. Suppose we have the proof tree described in Figure 5.1. In this case, the reasoner can potentially apply rule `prp-symp` (concerning symmetric properties in OWL) to derive some triples that are part of the second antecedent of rule `prp-spo1`.

However, in this case, Rule `prp-symp` will fire only if some of the subjects (i.e. the first component) of the triples part of *T(x, SPO, TYPE)* will also be the subject of *T(x, TYPE, SYM)*. Since both patterns are precalculated, we know beforehand all the possible 'x', and therefore we can immediately perform an intersection between the two sets to see whether this is actually the case. If there is an intersection, then the reasoner proceeds executing rule `prp-symp`, otherwise it can skip its execution since it will never fire.

It is very unlikely that the same property appears in all the terminological patterns, therefore an information passing strategy that is based on the precalculated triple patterns is very effective in significantly reducing the tree size and hence improve the performance.

In the following section, we will focus on the prematerialization phase, which purpose is to calculate the results for these subqueries in order to maintain our approach complete.

5.3 Hybrid Reasoning: Pre-Materialization

Before the user can query the knowledge base, our approach relies on a prematerialization phase where we calculate some subqueries so that during query time our backward-chaining algorithm is able to infer the entire derivation. We first formalize and discuss the pre-materialization algorithm and then we will show that suppressing the evaluation of pre-materialized subqueries leads to the same query answers that can be inferred with the original Database and the original program.

5.3.1 Pre-Materialization algorithm

Let $\mathfrak{I}$ be a database and P the program with a list L of atomic queries that are selected for pre-materialization.

We report the pre-materialization algorithm in Algorithm 13. In a first step (lines 1–3), the database is extended with auxiliary relations named S_Q for $Q \in L$. Each rule of the program P is rewritten (lines 5–12) by replacing every body atom $R_i(\bar{t}_i)$ with the query $S_Q(\bar{t}_i)$ if $R_i(\bar{t}_i) \sqsubseteq Q$, i.e. if the "answers" to $R_i(\bar{t}_i)$ are also yielded by Q. The new rule thus obtained is stored in a new program P'. In case the rule p contains no body atoms that need to be replaced, p is stored in P' as well.

In each repeat-loop pass (cf. lines 15–24), $\mathfrak{I}$ is extended in an external step (lines 17–19) with all answers for $Q \in L$, which are stored in the auxiliary relation $S_Q^{\mathfrak{I}}$. Since this is repeated between each derivation until no new answers

Algorithm 13 Overall algorithm of the precomputation procedure: L is a constant containing all queries that were selected for pre-materialization, $RuleSet$ is a constant containing a program P and $Database$ represents $\mathfrak{I}$.

```
1   for every Q ∈ L
2      introduce a new predicate symbol S_Q to Database
3   end for
4
5   for every rule p : R_0(t̄_0) ← R_1(t̄_1) ∧ ... ∧ R_n(t̄_n) in RuleSet
6      for every Q ∈ L
7         if R_i(t̄_i) ⊑ Q then
8            replace R_i(t̄_i) in p with S_Q(t̄_i)
9         end if
10     end for
11     add this (altered) rule to NewRuleSet
12  end for
13
14  Derivation := ∅
15  repeat
16     Database := Database ∪ Derivation
17     for every R(t̄) ∈ L
18        Perform S_R(t̄)(t̄) ← R(t̄) on Database
19     end for
20
21     for every Q in L
22        Derivation := Derivation ∪ main(Q) using NewRuleset as program on Database
23     end for
24  until Derivation ⊆ Database
```

for any $Q \in L$ are yielded, this is equivalent to adding $S_Q(\bar{t}) \leftarrow R(\bar{t})$ for each $R(\bar{t})$ with $Q = R(\bar{t})$ to P' directly[†]. This makes the procedure complete in the sense, that after termination of this algorithm S_Q contains all answers for the query Q in the full materialization of *Database* under *Ruleset*.

Example 5. *Take the altered program from Example 1 and add the appropriate $S_Q(Q) \leftarrow T(Q)$ to it. In this case we obtain*

$$\begin{aligned} T(a,p1,b) &\leftarrow S(p,SPO,p1) \wedge T(a,p,b) \\ T(x,SPO,y) &\leftarrow S(x,SPO,w) \wedge S(w,SPO,y) \\ S(x,SPO,y) &\leftarrow T(x,SPO,y) \end{aligned}$$

It is trivially clear, that this program yields for every Database exactly the same results for $T(x,SPO,y)$ as the original program

$$\begin{aligned} T(a,p1,b) &\leftarrow T(p,SPO,p1) \wedge T(a,p,b) \\ T(x,SPO,y) &\leftarrow T(x,SPO,w) \wedge T(w,SPO,y) \end{aligned}$$

[†]Formally, such rule would violate the Datalog condition that edb predicates do not appear in the head of rules. However, such violation in not influent for the purpose of our explanation.

Algorithm 13 terminates and is sound in the sense that after it has terminated, $S_{R(\bar{t})}(a_1, \ldots, a_n)$ is in $\mathfrak{I}$ whenever $R(a_1, \ldots, a_n)$ is an answer to $R(\bar{t})$ in the least fix-point model $P(\mathfrak{I})$ of P over $\mathfrak{I}$.

We consider the Algorithm 13 as complete if, after it has terminated, the auxiliary relations that we introduced contain all the triples that can be derived by launching the corresponding query with the original database $\mathfrak{I}$ and ruleset (P). More formally, this can be expressed as in the following proposition:

Proposition 3. *Algorithm 13 is complete in the sense that for the database $\mathfrak{I}_0$ which we obtain after Algorithm 13 has terminated $S_Q^{\mathfrak{I}_0} \supseteq Q^{P(\mathfrak{I})}$ for all $Q \in L$, i.e. every answer that could be derived from Q under P in $\mathfrak{I}$ is contained in $S_Q^{\mathfrak{I}_0}$.*

Proof. In order to proof the validity of this proposition, we assume for the sake of contradiction that Algorithm 13 were not complete: this means that no new element could be derived in line 22 from the current state of the database $\mathfrak{I}_0$ using the program P' (defined as *NewRuleset* in the pseudocode) but for some $Q \in L$, $main(Q)$ could derive another yet unknown fact from $\mathfrak{I}_0$ using the original program P. Let therefore $R(a_1, \ldots a_n)$ be the first yet underived answer for any $Q \in L$ which is derived under the original program P.

Line 18 guarantees that all $S_Q^{\mathfrak{I}_0} = Q^{\mathfrak{I}_0}$ and so program P' is at this stage indistinguishable from P. Hence $main(Q)$ must derive the fact $R(a_1, \ldots a_n)$ under P', as well. A contradiction! Since $S_Q^{\mathfrak{I}_0}$ never shrinks during the pre-materialization process, Algorithm 13 is complete. □

5.3.2 Reasoning with Pre-Materialized Predicates

In the previous section we have described the pre-materialization algorithm and demonstrated that it is able to calculate all the derivations to some chosen subqueries and store the results in some edb relations.

In order to verify the correctness of our approach, we still need to demonstrate that the procedure of replacing body atoms with auxiliary predicates that contain the full materialization of the body atom w.r.t. a given database, yields the same full materialization of the database as under the original program. The claim will be shown in all its generality explaining on the way, how the theoretical setting we draw up is connected to our specific case.

Let P be an arbitrary Datalog program and $\mathfrak{I}$ a database. We assume that $\mathfrak{I}$ has already been enriched with the results of the pre-materialization.

As an example, assume the binary relation $S^{\mathfrak{I}}$ contains all answer tuples of the query

$$query(x, y) \leftarrow T(x, SPO, y).$$

base.
under the program P.

From an abstract point of view we can define S as an *extensional database predicate* (edb) of P, i.e. it is not altered by P so that the *interpretation* $S^{\mathfrak{I}}$ of S under $\mathfrak{I}$ equals the interpretation $S^{P(\mathfrak{I})}$ of S under the *least fix-point model* $P(\mathfrak{I})$ of P over $\mathfrak{I}$.

Since S is an edb and therefore does not appear in the head of any rule of P, S cannot be unfolded and so the evaluation of S during the backward chaining process is reduced to a mere look-up in the database.

Such a replacement in the original program is harmless only if $\mathfrak{I}$ has been adequately enriched. Thus, the question arises which abstract conditions must be satisfied to allow such a replacement: In essence, we want that a rule fires under "almost the same" variable assignment as its replacement, which we formalize in the following two paragraphs.

Assume $R_0, \ldots, R_n$ are predicates of the program P. Let $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_n(\bar{t}_n)$ be a rule in P. The $\bar{t}_i = (t_{i,1}, \ldots t_{i,m_i})$ represent tuples of terms, where each term is either a variable or an element of the domain in $\mathfrak{I}$ and m_i equals to the arity of R_i for all $i \in \{1, \ldots, n\}$.

We define two queries, one being the body of the rule and one being the body of the rule where one body atom $R_i(\bar{t}_i)$ is replaced by S: Let $\bar{z} := \bar{t}_0 \cdot \bar{t}_1 \cdots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \cdots \bar{t}_n$, i.e. the concatenation of all tuples except $\bar{t}_i$ and let $\bar{t}$ be some arbitrary tuple.

$$\begin{aligned} q_0(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_i(\bar{t}_i) \wedge \ldots \wedge R_n(\bar{t}_n) \\ q_1(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge S(\bar{t}) \quad \wedge \ldots \wedge R_n(\bar{t}_n) \end{aligned} \qquad (*)$$

Now, the rule and its replacement fire under *almost the same variable assignment* if $q_0^{P(\mathfrak{I})} = q_1^{P(\mathfrak{I})}$, i.e. q_0 and q_1 yield the same answers under P in $\mathfrak{I}$. We see, that it is "almost the same" variable assignment, as we do not require variable assignments to coincide on $\bar{t}_i$ and $\bar{t}$. In this way we do not require, e.g., $S^{\mathfrak{I}} = R_i^{P(\mathfrak{I})}$. S is merely required to contain the *necessary* information. This is important, if we want to apply the substitution to rdf-triples, where we lack distinguished predicate names:

Example 6. *Since there is only one generic predicate symbol T, requiring $S^{\mathfrak{I}} = T^{P(\mathfrak{I})}$ would mean that S contains the complete materialization of $\mathfrak{I}$ under P which would render our approach obsolete.*

Also notice, that it is not sufficient to merely require $q_0(\bar{t}_0)^{P(\mathfrak{I})} = q_1(\bar{t}_0)^{P(\mathfrak{I})}$, i.e. that both queries yield the same answer tuples $\bar{t}_0$ under $P(\mathfrak{I})$, as the following example shows.

Example 7. *Let the program P which computes the transitive closure of R_0 in R_1 consist of the two rules:*

$$\begin{aligned} R_1(x,z) &\leftarrow R_1(x,y) \wedge R_0(y,z) \\ R_1(x,y) &\leftarrow R_0(x,y) \end{aligned}$$

Consider database $\mathfrak{I}$ with $R_0^{\mathfrak{I}} := \{(a,b),(b,c),(b,b),(c,c)\}$. In the least fix-point model $P(\mathfrak{I})$ of P we expect $R_1^{P(\mathfrak{I})} = \{(a,b),(b,c),(a,c),(b,b),(c,c)\}$. Let S have the interpretation $S^{\mathfrak{I}} = \{(b,b),(c,c)\}$. Since $R_1^{P(\mathfrak{I})}$ is the transitive closure, the following two queries deliver the same answer tuples under $P(\mathfrak{I})$:

$$\begin{aligned} q_0(x,z) &\leftarrow R_1(x,y) \wedge R_0(y,z) \\ q_1(x,z) &\leftarrow R_1(x,y) \wedge S(y,z) \end{aligned}$$

Yet the program P'

$$\begin{aligned} R_1(x,z) &\leftarrow R_1(x,y) \wedge S(y,z) \\ R_1(x,y) &\leftarrow R_0(x,y) \end{aligned}$$

will not compute the transitive closure of R_0 in R_1 because $R_1^{P'(\mathfrak{I})} = \{(a,b),(b,c),(b,b),(c,c))\}$.

We shall now show that substituting a body atom $R_i(\bar{t}_i)$ by $S(\bar{t})$ under the condition that the queries in $(*)$ yield the same answer tuples under $P(\mathfrak{I})$, generates the same least fix-point:

Proposition 4. *Let P' be the program P where the rule*
$R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_i(\bar{t}_i) \wedge \ldots \wedge R_n(\bar{t}_n) \in P$
has, for some tuple $\bar{t}$ and edb S, been replaced by
$R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge S(\bar{t}) \wedge \ldots \wedge R_n(\bar{t}_n)$.
Let q_0 and q_1 be defined as in $(*)$.

$$\text{If } q_0^{P(\mathfrak{I})} = q_1^{P(\mathfrak{I})} \text{ then } P(\mathfrak{I}) = P'(\mathfrak{I}).$$

Proof. In order to show the implication we assume $q_0(\bar{z})^{P(\mathfrak{I})} = q_1(\bar{z})^{P(\mathfrak{I})}$. Let T_P and $T_{P'}$ be the immediate consequence operators (mentioned on page 93) for each program. We show for all $k < \omega$ that if $Q(a_1, \ldots a_m) \in T_P^k(\mathfrak{I})$ then there is an $\ell < \omega$ such that $Q(a_1, \ldots a_m) \in T_{P'}^{\ell}(\mathfrak{I})$ and vice versa. Since we

start out from the same database $\mathfrak{I}$ we have $T_P^0(\mathfrak{I}) = T_{P'}^0(\mathfrak{I})$ which settles the base case.

Let R be an intensional predicate of P, i.e. it appears in some rule head in P. If $R(a_1, \dots a_m) \in T_P^{k+1}(\mathfrak{I})$ then either $R(a_1, \dots a_m) \in T_P^0(\mathfrak{I})$ and we are done or there is some rule $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ and some variable assignment β such that $\beta(\bar{t}_0) = (a_1 \dots, a_m)$ and $R_j(\beta(\bar{t}_j)) \in T_P^k(\mathfrak{I})$ for all $j \in \{1, \dots, n\}$.

If $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n) \in P'$, i.e. none of its body atoms where substituted, the induction hypothesis shows for each $j \in \{1, \dots, n\}$ that we can find $\ell_j < \omega$ such that $R_j(\beta(\bar{t}_j)) \in T_{P'}^{\ell_j}(\mathfrak{I})$. Let $\ell_0 := \max\{1\} \cup \{\ell_j \mid 1 \leq j \leq n\}$. Notice, that we add $\{1\}$ for the case where the rule body was empty. In any case, we have $R_j(\beta(\bar{t}_j)) \in T_{P'}^{\ell_0}(\mathfrak{I})$ for all $j \in \{1, \dots, n\}$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(\bar{t}_0)) \in T_{P'}^{\ell}(\mathfrak{I})$.

If $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n) \notin P'$ it is the rule where $R_i(\bar{t}_i)$ has been substituted with $S(\bar{t})$. For the assignment β we now know $\beta(\bar{t}_0 \cdot \bar{t}_1 \cdots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \cdots \bar{t}_n) \in q_0(\bar{z})^{P(\mathfrak{I})}$. Since $q_0(\bar{z})^{P(\mathfrak{I})} = q_1(\bar{z})^{P(\mathfrak{I})}$ we know that there is some assignment β', which coincides with β on $(\bar{t}_0 \cdot \bar{t}_1 \cdots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \cdots \bar{t}_n)$ such that $\beta'(\bar{t}) \in S^{P(\mathfrak{I})}$.

Hence $R_j(\beta'(\bar{t}_j)) \in T_P^k(\mathfrak{I})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$ and $S(\beta'(\bar{t})) \in T_P^0(\mathfrak{I})$ since S is an edb predicate. The induction hypothesis yields some $\ell_j < \omega$ for each $j \in \{1, \dots, n\} \setminus \{i\}$ such that $R_j(\beta'(\bar{t}_j)) \in T_{P'}^{\ell_j}(\mathfrak{I})$. Let $\ell_0 := \max\{0\} \cup \{\ell_j \mid 1 \leq j \leq n \text{ and } j \neq i\}$, then $R_j(\beta'(\bar{t}_j)) \in T_{P'}^{\ell_0}(\mathfrak{I})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$ and $S(\beta'(\bar{t})) \in T_{P'}^0(\mathfrak{I})$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta'(\bar{t}_0)) \in T_{P'}^{\ell}(\mathfrak{I})$. As β coincides with β' also on $\bar{t}_0$, i.e. $\beta'(\bar{t}_0) = (a_0, \dots, a_m)$, we have in particular $R(a_0, \dots, a_m) \in T_{P'}^{\ell}(\mathfrak{I})$.

This shows that for all predicates Q we have $Q^{P(\mathfrak{I})} \subseteq Q^{P'(\mathfrak{I})}$. For the converse we merely show the case of the substituted rule: Assume $R(a_1, \dots a_m) \in T_{P'}^{k+1}(\mathfrak{I})$ and there is an assignment β' such that $\beta'(\bar{t}_0) = (a_1, \dots a_m)$ and $R_j(\beta'(\bar{t}_j)) \in T_{P'}^k(\mathfrak{I})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$ as well as $\beta'(\bar{t}) \in S^{P'(\mathfrak{I})}$.

The induction hypothesis yields for each $j \in \{1, \dots, n\} \setminus \{i\}$ some $\ell_j < \omega$ with $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_j}(\mathfrak{I})$. Since S is an edb predicate for P, we have $S(\beta'(\bar{t})) \in T_P^0(\mathfrak{I})$. Hence for $\ell_0 := \max\{0\} \cup \{\ell_j \mid 1 \leq j \leq n \text{ and } j \neq i\}$ we have $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_0}(\mathfrak{I})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$ and $S(\beta'(\bar{t})) \in T_P^0(\mathfrak{I})$.

This implies $\beta'(\bar{t}_0 \cdot \bar{t}_1 \cdots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \cdots \bar{t}_n) \in q_1(\bar{z})^{P(\mathfrak{I})}$ and since $q_0(\bar{z})^{P(\mathfrak{I})} = q_1(\bar{z})^{P(\mathfrak{I})}$ there is an assignment β coinciding on $(\bar{t}_0 \cdot \bar{t}_1 \cdots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \cdots \bar{t}_n)$ with β' such that $R_i(\beta(\bar{t}_i)) \in T_P^{j_0}(\mathfrak{I})$ for some $j_0 < \omega$. Let $\ell_1 := \max\{\ell_0, j_0\}$ then $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_1}(\mathfrak{I})$ for all $j \in \{1, \dots, n\}$. Since all premises of the rule $R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ are satisfied, there is some $\ell < \omega$ such

that $R_0(\beta(\bar{t}_0)) \in T_P^\ell(\mathfrak{I})$, which shows, as β coincides on t_0 with β' that $R(a_0, \ldots, a_m) \in T_P^\ell(\mathfrak{I})$.

Together with $Q^{P(\mathfrak{I})} \subseteq Q^{P'(\mathfrak{I})}$ this shows $Q^{P(\mathfrak{I})} = Q^{P'(\mathfrak{I})}$ for all predicate names Q and hence that $P(\mathfrak{I}) = P'(\mathfrak{I})$. □

It now becomes clear, how Algorithm 13 and Proposition 4 fit together: For a given database $\mathfrak{I}$ and a list of atomic queries L, Algorithm 2 computes for each $Q \in L$ the query answers under the program P, which are stored in the relation $S_Q^{\mathfrak{I}}$. These S_Q are edbs for P.

Let now $r : R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_n(\bar{t}_n)$ be a rule in this program and $Q \in L$ an atomic query s.t. $R_i(\bar{t}_i) \sqsubseteq Q$, then $Q = R_i(\bar{t})$ such that $\bar{t}_i \sqsubseteq \bar{t}$ by definition of $\sqsubseteq$. Correctness of Algorithm 13 yields $R(\bar{t}_i)^{P(\mathfrak{I})} = S_Q(\bar{t}_i)^{P(\mathfrak{I})}$ and hence that $q_0(\bar{z})^{P(\mathfrak{I})} = q_1(\bar{z})^{P(\mathfrak{I})}$ where $\bar{z} = \bar{t}_0 \cdots \bar{t}_n$ and

$$\begin{aligned} q_0(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_i(\bar{t}_i) \wedge \ldots \wedge R_n(\bar{t}_n) \\ q_1(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge S_Q(\bar{t}_i) \wedge \ldots \wedge R_n(\bar{t}_n) \end{aligned}$$

Proposition 4 guarantees that the substitution of $R_i(\bar{t}_i)$ by $S_Q(\bar{t}_i)$ in rule r is harmless w.r.t. $\mathfrak{I}$. By applying this argument iteratively, one eventually obtains a program P' in which all pre-computed atoms have been replaced and which yields the same materialization for $\mathfrak{I}$ as P.

In the following section we shall apply this rewriting to the OWL RL rule set.

5.4 Hybrid reasoning for OWL RL

In the previous sections we described the two main components of our method which consists of the backward-chaining algorithm used to retrieve the inference and the pre-materialization procedure which ensures the completeness of our approach.

We will now discuss the implementation of the OWL RL rules using our approach. The official OWL RL rule set contains 78 rules, for which the reader is referred to the official document overview [92]. With some selected examples from [92] we will illustrate some key features of our algorithm.

Initial assumptions. First of all, we exclude some rules from our discussion and implementation for various reasons. These are:

- All the rules whose purpose is to derive an inconsistency, i.e. rules with predicate *false* in the head of the rule. We do not consider them because our purpose is to derive new triples rather than identify an inconsistency;

OWL RL	*pD**	*RDFS*	`(?X rdfs:subPropertyOf ?Y)`
			`(?X rdfs:subClassOf ?Y)`
			`(?X rdfs:domain ?Y)`
			`(?X rdfs:range ?Y)`
			`(?P rdf:type owl:FunctionalProperty)`
			`(?X owl:allValuesFrom ?Y)`
			`(?P rdf:type owl:InverseFunctionalProperty)`
			`(?X owl:inverseOf ?Y)`
			`(?P rdf:type owl:TransitiveProperty)`
			`(?X rdf:type owl:Class)`
			`(?P rdf:type owl:SymmetricProperty)`
			`(?X rdf:type owl:Property)`
			`(?X owl:equivalentClass ?Y)`
			`(?X owl:onProperty ?Y)`
			`(?X owl:hasValue ?Y)`
			`(?X owl:equivalentProperty ?Y)`
			`(?X owl:someValuesFrom ?Y)`
			`(?X owl:propertyAxiom ?Y)`
			`(?X owl:hasKey ?Y)`
			`(?X owl:intersectionOf ?Y)`
			`(?X owl:unionOf ?Y)`
			`(?X owl:oneOf ?Y)`
			`(?X owl:maxCard 1)`
			`(?X owl:maxQCar 1)`
			`(?X owl:onClass ?Y)`
			`(?X rdf:type owl:Class)`
			`(?X rdf:type DataTypeProp)`
			`(?X rdf:type ObjTypeProp)`

Table 5.2: Triple patterns that are precalculated considering the OWL RL rules.

– All the rules which have an empty body. These rules cannot be triggered during the unfolding process of backward-chaining. These rules include the ones who encode the semantics of datatypes, therefore our implementation does not support datatypes;

– The rules that exploit the *owl:sameAs* transitivity and symmetry‡. These rules require a computation that is too expensive to perform at query time since they can be virtually applied to every single term of the triples. These rules can be implemented by computing the *sameAs* closure and maintaining a consolidation table.§

In our approach we decided to pre-materialize all triple patterns that are used to retrieve "schema" triples, also referred to as the terminological triples.

‡The list of these rules is reported in Table 4 of [92].
§This procedure is explained in detail in Chapter 2.

In Table 5.2 we report the list of the patterns that are pre-materialized using our method described in Section 5.3.

Singling out exactly those triple patterns from Table 5.2 is motivated by the empirical observation that:

- they appear in many of the OWL rules;
- their answer sets are very small compared to the entire input;
- their answer sets are not as frequently updated as the rest of the data.

These characteristics make the set of inferred schema triples the ideal candidate to be pre-materialized. All rules which have a pre-materialized pattern amongst their body atoms, are substituted replacing the pre-materialized pattern with its corresponding auxiliary relation as justified by Proposition 4. This affects 25 rules out of 78 and hence reduces reasoning considerably.

After the pre-materialization procedure is completed, each rule which has a pre-materialized pattern in its head can be reduced to a mere look-up:

Example 8. *Consider (scm-sco) from Table 9 in [92]:*

$$T(x, SCO, z) \leftarrow T(x, SCO, y) \wedge T(y, SCO, z)$$

can be replaced by Proposition 4

$$T(x, SCO, z) \leftarrow S_{sco}(x, SCO, y) \wedge S_{sco}(y, SCO, z)$$

where all answers to `T(?c₁ SCO ?c₂)` *are contained in* $S^{\mathfrak{I}}_{sco}$. *Then* $T(x, SCO, y)$ *can be replaced by* $\top$ *the constant for true, again using Proposition 4, and finally obtaining*

$$T(x, SCO, z) \leftarrow S_{sco}(x, SCO, z)$$

This removes a further 30 rules from unfolding.

On the implementation of RDF lists. Some of the inference rules in the OWL RL rule set use RDF lists to define a variable number of antecedents. The RDF lists cannot be represented in Datalog in a straightforward way since they rely on *rdf:first* and *rdf:next* triples to represent the elements of the list. Therefore, they need to be processed differently.

In our implementation, at every step of the pre-materialization procedure, we launch two additional queries to retrieve all the (inferred and explicit) *rdf:first* and *rdf:rest* triples with the purpose of construct such lists. Once we have collected them, we perform a join with the other schema triples, and determine the sequence of elements by repetitively joining the *rdf:first* and

rdf:rest triples. After this operation is completed, from the point of view of the Datalog program RDF lists appear as simple list of elements and are used according to the rule logics.

5.4.1 Detecting duplicate derivation in OWL RL

Since the OWL RL fragment consists of a large number of rules, there is a high possibility that the proof tree contains branches that lead to the same derivation. Detecting and avoiding the execution of these branches is essential in order to reduce the computation.

After empirical analysis on some example queries we determined that there are two types of sources in the generation of duplicates. The first comes from the nature of the ruleset. The second comes from the input data.

First type of duplicates source. The most prominent example of generation of duplicates of the first type is represented by the *symmetric* rules which have the same structure but have the variables positioned at different locations. We refer with rule names and tables in the following list to the OWL RL ruleset in [92]:

`prp-eqp1`	and	`prp-eqp2`	from Table 5
`cax-eqc1`	and	`cax-eqc2`	from Table 7
`prp-inv1`	and	`prp-inv2`	from Table 5.

We will now analyze each of these three cases below.

Let $S^{\mathfrak{I}}_{eqp}$ be the pre-materialization of the triple pattern $\mathtt{T(x,EQP,y)}$. Rules `scm-eqp1` and `scm-eqp2` render $S^{\mathfrak{I}}_{eqp}$ symmetric. Hence

$$q(x,p_2,y) \leftarrow T(x,p_1,y) \wedge S_{eqp}(p_1,EQP,p_2)$$

yields the same results under $P(\mathfrak{I})$ as

$$q(x,p_1,y) \leftarrow T(x,p_2,y) \wedge S_{eqp}(p_2,EQP,p_1)$$

and Proposition 4 yields that `scm-eqp2` can be replaced by `scm-eqp1`, effectively deleting `scm-eqp2` from the ruleset.

Similarly, rules `scm-eqc1` and `scm-eqc2` render the results of the pre-materialized query $\mathtt{T(x,EQC,y)}$, symmetric.

In contrast, the pre-materialized query $\mathtt{T(x,INV,y)}$ is not symmetric. However, we can first observe that Proposition 4 allows to replace the rules by

$$T(x,p,y) \leftarrow T(x,q,y) \wedge S_{inv}(p,INV,q)$$
$$T(x,p,y) \leftarrow T(x,q,y) \wedge S_{inv}(q,INV,p)$$

which defuses the idb atom $T(q, INV, p)$ into the harmless edb S_{inv}, for which $S_{inv}^{\mathfrak{I}}$ contains all answers to the query $\mathtt{T(x, INV, y)}$. Let this new program be called P'. Further, let P'' be the program where both rules have been replaced by

$$T(x, p, y) \leftarrow T(x, q, y) \wedge S'_{inv}(p, INV, q)$$

with $S'^{\mathfrak{I}}_{inv}$ being the symmetric closure of $S_{inv}^{\mathfrak{I}}$. It is now not difficult to see, that every model of P' is a model of P'' and vice versa. In particular the least fix-point model of $P(\mathfrak{I})$ is equal to the least fix-point model $P''(\mathfrak{I})$. Hence we can replace `prp-inv1` and `prp-inv2` by one rule under the condition that we pre-materialize the symmetric closure of $\mathtt{T(x, INV, y)}$.

Second type of duplicates source. The second type of duplicate generations comes from the input data which might contain some triples that make the application of two different rules perfectly equivalent.

We have identified an example of such a case in the Linked Life Dataset, that is one realistic dataset that we used to evaluate our approach. In this dataset there is the triple:

T(SCO, TYPE, TRANS)

which states that the *subClassOf* predicate is transitive.

In this case, during the precomputation phase the query *T(a, SCO, b)* will be launched several times, and each time the reasoner will trigger the application of both the rules `scm-sco` and `prp-trp`.

However, since the application of these two rules will lead to the same derivation, such computation is redundant and inefficient. Therefore, to detect such cases we can apply a special algorithm when the system is starting up and it is initializing the ruleset. A complete description of this algorithm is outside the scope of this chapter and we will simply illustrate the main idea behind it.

Basically, this algorithm compares each rule with all the others in order to identify under which conditions the two will produce the same output to a given query. For example, the rules `scm-sco` and `prp-trp` will produce the same derivation if (i) the input contains the triple *T(SCO, TYPE, TRANS)* and if (ii) there is a query with *SCO* as a predicate.

In order to verify this is the case, the algorithm checks whether the triple *T(SCO, TYPE, TRANS)* exists in the input and there is a matching on the position of the variables in the two rules (if one rule contains more variables than the other, then the algorithm will substitute the corresponding terms). If such matching exists, then the two rules are equivalent. In our example,

the algorithm will find out that the rule `prp-trp` is equivalent to `scm-sco` if we replace *?p* with *SCO*. Therefore, if there is an input query with *SCO* as predicate, the system will execute only one of the two rules, avoiding in this way a duplicated derivation.

5.5 Evaluation

We have implemented our approach in a Java prototype that we called *QueryPIE* and we evaluated the performance using one machine of the DAS-4 cluster¶, which is equipped with a dual Intel E5620 quad core CPU of 2.4 GHz, 24 GB of memory and 2 hard disks of 1 TB each configured in RAID-0 mode.

We used two datasets as input. LUBM [30] which is one of the most popular benchmarks for OWL reasoning and LLD (Linked Life Data)‖, which is a curated collection of real-world datasets in the bioinformatics domain.

LUBM allows to generate datasets of different sizes. For our experiments we generated a dataset of 10 billion triples (which corresponds to the generation of 80000 LUBM universities). The Linked Life Data dataset consists of about 5 billion triples. Both datasets were compressed using the procedure described in [87].

We organize this section as follows. First, in Section 5.5.1 we will report a set of experiments to evaluate the performance of the pre-materialization phase. Next, in Section 5.5.2 we will focus on the performance of the backward-chaining approach and analyze its performance on some example queries. Finally, in Section 5.5.3, we present a more general discussion on the results that we obtained.

5.5.1 Performance of the pre-materialization algorithm

We launched the pre-materialization algorithm on the two datasets to measure the reasoning time necessary to perform the partial closure. The results are reported in the second column of Table 5.3. Our prototype performs joins between the pre-materialized patterns when it loads the rules in memory, therefore, we have also included the startup time along with the query runtimes to provide a fair estimate of the time requested for the reasoning.

From the results, we notice that the pre-materialization is about three orders of magnitude faster for the LUBM dataset than for LLD. The reason

¶`http://www.cs.vu.nl/das4`

‖`http://www.linkedlifedata.com/`

Dataset	Reasoning time		N. iterations	N. derived triples
	Our approach	Full materialization		
LUBM	1s	4d4h16m	4	390
LLD	16m	5d10h45m	7	10 millions

Table 5.3: Execution time of the pre-materialization algorithm compared to a full closure.

Pattern	Dataset	Query
1	LUBM	?x ?y <http://www.Dept0.../GraduateCourse0>
2	LUBM	?x <lubm:subOrganizationOf> <http://www.University0.edu>
3	LUBM	<...GraduateStudent124> <lubm:degreeFrom> <...niversity114>
4	LUBM	?x ?y <http://www.Dept0.../AssistantProfessor0>
5	LUBM	?x <lubm:memberOf> <http://www.Dept0.Universitity0.edu>
6	LUBM	?x <rdf:type> <lubm:Department>
7	LLD	?x ?y <lifeskim:mentions>
8	LLD	?x <lifeskim:mentions> <.../umls/id/C0439994>
9	LLD	<.../resource/pubmed/id/15964627> ?x ?y
10	LLD	?x ?y <http://purl.uniprot.org/go/0006952>
11	LLD	?x ?y <http://linkedlifedata.com/resource/umls/id/C0439994>
12	LLD	?x <http://www.biopax.org/.../biopax-level2.owl#NAME> ?y

Table 5.4: List of example queries

behind this difference is that the ontology of LUBM requires much less reasoning than the one of LLD in order to be pre-materialized. In fact, in the first case the pre-materialization algorithm has derived about 390 triples and needing four iterations to reach a fix point. On LLD the pre-materialization required 7 iterations and returned about 10 million triples.

We intend to compare the cost of performing the partial closure against the cost of a full materialization, which is currently considered as the state of the art in the field of large scale OWL reasoning. However, to the best of our knowledge there is no approach described in literature which supports the OWL RL fragment and which is able to scale to the input size that we consider.

The closest approach we can use for a comparison is the approach that we presented in Chapter 2 which we used to demonstrate OWL reasoning up to

the $pD*$ fragment to a hundred billion triples. Since this approach uses the MapReduce programming model, an execution on a single machine would be suboptimal. Therefore, we launched it using eight machines and multiplied the execution time accordingly to estimate the runtime on one machine (such estimation is in line with the performance of WebPIE which has shown linear scalability, as described in section 2.4.4).

The runtime of the complete materialization performed with this method is reported in the third column of Table 5.3. We notice that in both cases a complete materialization requires between four and five days against the seconds or minutes required for hybrid reasoning. This comparison clearly illustrates the advantage of our approach in terms of pre-materialization cost. However, such advantage comes at a price: while after a complete materialization reasoning is no longer needed, in our case we still have to perform some inference at query time. The impact of this operation on the query-time performance is analyzed in the next section.

5.5.2 Performance of the reasoning at query time

In order to analyze the performance of reasoning at query time, we launched some example queries after we computed the closure using our backward-chaining algorithm to retrieve the results. To this purpose, we selected six example queries for both the LUBM and LLD datasets and report them in Table 5.4.

While LUBM provides an official set of queries for benchmarking, unfortunately there is no official set of queries that can be used for benchmarking the performance on the LLD dataset. Therefore, we took some queries that are reported in the official page of the LLD dataset and modified them so that they could trigger different types of reasoning.

These queries were selected according to the following criteria:

- *Number of results:* We selected queries that return a number of results that varies from no results to a large set of triples;

- *Reasoning complexity:* Some queries in our example set require no reasoning to be answered, in contrast other queries generate a very large proof-tree;

- *Amount of data processed:* In order to answer a query, the system might need to access and process a large set of data. We selected queries that read and process a variable amount of data to verify the impact of I/O on the overall performance.

Query	Runtime (ms)		Processed Triples		I/O access	
	Cold	Warm	Total	Output	# lookups	MB
1	60.43	6.39	5	5	43	8
2	1099.28	129.31	463	239	12	205
3	49.18	6	3	1	18	5
4	73.06	11.17	37	29	86	8
5	118.71	13.97	1480	719	18	8
6	4026.27	2590.27	1599987	1599987	2	12
7	228.26	214.57	0	0	670	23
8	23.74	6.29	4466	4466	1	4
9	7064.04	609.4	140	128	3540	105
10	2535.38	1103.48	28446	26860	14372	337
11	2613.37	1883.14	8546	4504	15128	64
12	2334.70	2059.20	1187944	1187944	1	10

Table 5.5: Runtime of the queries in Table 5.4 on the LUBM and LLD datasets

We performed a number of experiments to analyze three aspects of the performance of our algorithm during query time: the *absolute response time*, the *reduction of the proof-tree*, and the *overhead induced by reasoning during query-time*. Each of these aspects is analyzed below.

Absolute response time

We report in Table 5.5 the execution time obtained launching the selected example queries in Table 5.4. In the second and third columns we report both the cold and warm runtime**. With cold runtime we identify the runtime that is obtained by launching the query right after the system has started. Since the data is stored on disk, with the cold runtime we also measure the time to read the data from disk. On the other side, the warm runtime measures the average response time of launching the same query thirty more times. Because during such execution the data is already cached in memory and the Java VM has already initialized the internal data structures, the warm runtime is significantly faster than the cold one.

The fourth and fifth column, respectively, report the total number of derivations that were inferred during the execution of the query, and the number of

**Notice that the reported runtime does not include the time required to compress/decompress the numerical terms to their string counterpart.

triples returned to the user.

The sixth and seventh column report the number of data lookups required to answer the query and the amount of data that is read from disk. These two numbers are important to estimate the impact of reasoning at query time. While one query without reasoning requires only one data lookup, in our case the reasoning algorithm might require to access the database multiple times. For example, in order to answer query 11 the program had access to the data indices about 15000 times.

From the results reported in Table 5.5 we can make several considerations. First, we notice that the cold runtime is in general significantly higher than the warm runtime between one and two orders of magnitude. This is primarily due to expensive cost of the I/O access to disk especially because reasoning requires to read at different locations of the data indices, and therefore the system is required to read several blocks of the B-Tree from the disk. For most of the queries, the I/O access dominates the execution time. The worst case is represented by query 10 where the program reads from disk about 337 MB of data. From these results we conclude that the performance of the program in case the data is stored on disk is essentially I/O bounded. After the data is loaded in memory, the execution time drops by about one order of magnitude on average and the performance becomes CPU bounded.

Another factor that impacts the performance is the number of the inferred triples that are calculated during the execution of the query. In fact, we notice that absolute performance is lower in case a large number of triples is either inferred or retrieved from the database. The behavior is due to the fact that the algorithm needs to temporarily store these triples as it must consider them in each repeat-loop pass until the closure is reached. This means that these triples must be stored and indexed to be retrieved during the following iterations and the response time consequently increases.

Summarizing our analysis, we make the following conclusions: (i) the runtime is influenced by several factors among which the most prominent is the amount of I/O access that is requested to answer the query (this number is proportional to the size of the proof tree) and the number of derivations produced. (ii) There is a large difference in the runtime observed in our experiments. In the worst case the absolute runtime is in the range of few seconds, while in the best cases the performance is in the order of dozens of milliseconds. However, even in the worst case the system allows an interactive usage since few seconds are acceptable in most scenarios.

In Section 5.5.2 we will compare such response times with the ones without reasoning in order to have a better overview of the overall performance and understand what is the overhead induced by reasoning at query time.

Query	# Leaves proof-tree		Reduction ratio
	Without precomp.	Our approach	
1	16	4	4.0
2	2	1	2.0
3	12	3	4.0
5	26	7	3.7

Table 5.6: Estimation of the reduction of the proof tree caused by the pre-materialization algorithm.

Q.	Only Lookup		RDFS		pD*		OWL RL	
	No Ins	Ins	No Ins	Ins	No Ins	Ins	No Ins	Ins
1	0.81	0.83	1.88	1.79	5.4	6.13	6.39	5.89
2	0.82	1.51	1.56	2.83	128.78	131.05	129.31	138.53
3	0.82	0.83	3.55	2.72	5.50	4.51	6	4.83
4	0.88	0.94	2.01	2.32	10.06	9.48	11.17	10.63
5	1.5	1.61	7.01	4.95	13.58	10.52	13.97	10.8
6	405.42	418.38	2605.68	2630.08	2608.20	2619.17	2590.27	2618.66
7	0.77	0.79	176.19	1.26	203.23	17.93	214.57	16.78
8	1.96	1.89	6.23	6.34	6.39	6.46	6.29	6.36
9	0.84	0.90	262.7	46.53	590.34	277.55	609.4	277.02
10	7.90	7.29	212.57	115.16	903.31	814.95	1103.48	1053.33
11	1.85	1.93	200.55	8.35	1695.73	1468	1883.14	1529.64
12	338.14	337.41	2129.49	2044.34	2055.02	2077.55	2059.2	2062.65

Table 5.7: Runtime (in ms.) of the example queries changing the ruleset.

Reduction of the proof tree

The backward-chaining algorithm and more in general our approach relies on the pre-materialization of some selected queries which serve a variety of purposes such as performing efficient sideways information passing or excluding rules that derive duplicates. Another advantage of performing the pre-materialization is that it reduces the size of the proof tree during query-time.

In this section, we will evaluate the effective reduction in terms of the size of the proof tree obtained by avoiding performing inference on the pre-materialized patterns.

However, since the method presented in this chapter is embedded in the implementation of our prototype, and since the optimizations introduced are

crucial to its execution, we cannot disable them. To overcome this problem, we have manually analyzed the execution of the LUBM queries with our prototype on a much smaller dataset and manually constructed the proof tree without pre-materialization (note that we excluded queries 4 and 6 since in these cases reasoning did not contribute to derive new answers). In principle, for each query, we identified the rules that produce some derivations and for each pre-materialized query in their body, we added the corresponding branch that was generated when that query was calculated during the pre-materialization phase.

We report the results of such analysis in Table 5.6. The last column reports the obtained reduction ratio and shows that the number of leaves shrinks between two and four times due to our pre-materialization. The results of this method of evaluation must be seen as an underestimate, because we could not deactivate all the optimizations, and therefore in reality the gain is even higher than the one calculated. Nevertheless, this shows that our pre-calculation is indeed effective. For a very small cost in both data space and upfront computation time, we substantially reduce the proof-tree. Apparently, the pre-materialization precisely captures small amounts of inferences that contribute substantially to the reasoning costs because they are being used very often.

Overhead of reasoning during query-time

While we are able to significantly reduce the size of the proof-tree and apply other optimizations to further reduce the computation, we still have to perform some reasoning during the execution of a query. It is important to evaluate what the cost for the remaining reasoning is when we compare our approach to a full-materialization approach (which is currently the de-facto technique for large scale reasoning), where a large pre-materialization is performed so that during query time reasoning is avoided altogether.

To this end, we launched a number of experiments activating different types of reasoning at query time and report the results in Table 5.7.

We proceeded as follows: we first launch the queries, deactivating all rules at query time, and state their execution time in the first column of the table (the title "No Ins." indicates no insertion). We then reissued the queries activating only the RDFS rules (in the third column), then the $pD*$ rules and finally the OWL RL ones.

The results reported under the "Ins." columns were calculated differently. In fact, in the previous experiments the number of retrieved results for a specific query might differ because we changed the rule set and this can influence the general performance. To maintain the number of results constant, we have

repeated the same experiment adding to the knowledge base all the possible results so that even if reasoning is not activated the same number of results is retrieved ("Ins." means insertion).

From the results presented in the table, we notice that the response time progressively increases as we include more rules. Such a behavior is clearly expected since more computation must be performed as we add new rules. However, in some cases (like query 12) there is a significant difference even if the query does not require the application of any rule. The difference is due to the cost of storing the results during the query execution to ensure the completeness of the backward-chaining algorithm. This operation is clearly a non-negligible contributor to the overall performance.

We can compare the response times reported in the third column with the ones of the penultimate column to compare the performance of the reasoning at query time of our approach against traditional full materialization. In fact, because the input data already contains the whole derivation, a single lookup can be used to estimate the cost that we would have to pay if all the inferences were pre-materialized beforehand. From the results we notice that on average the response time is between one and three orders of magnitude slower. In case the query needs to process and/or return many triples, the difference is certainly significant. However, the response time is still in the order of the hundreds of milliseconds and therefore, from the user perspective, the difference is less noticeable and more easily tolerated especially considering that a large precomputation phase is no longer needed.

5.5.3 Discussion

In our evaluation, we chose to evaluate our method using the most common and large-scale datasets currently available in order to evaluate how hybrid reasoning would perform on *current* data and *realistic* queries.

The measurements that we report have shown that large scale OWL RL reasoning is indeed possible, even on a relatively modest computer architecture. However, we must point out that these datasets do not (yet) use all the features introduced with the OWL 2 language and, to the best of our knowledge, there is no large-scale benchmark that extensively uses these new features.

Therefore, there is a remaining open question on what the performance would be on an input that exploits all the features of the OWL 2 language. While such problem is beyond the scope of this chapter, we can make some considerations by looking at the experiments here presented.

First of all, our approach most likely would not able to guarantee the same response time in the worst-case scenario. This is not particularly due to a

limitation of our method, but rather to the high computational complexity intrinsically required by reasoning.

However, even without looking at the complete worst-case scenario (which is very unlikely to happen in practice), there can be other cases where the performance could be significantly worse. In our experiments, we noticed that as the size of the proof tree increases, so does the potential derivation of duplicates due to the potential higher number of combinations. In Section 5.4.1, we tackled this problem by proposing some initial algorithms to limit the number of duplicates. However, our work in this respect is still initial and further research on this particular aspect might become necessary in order to scale not only in terms of input size but also in terms of reasoning complexity.

Summarizing, we observe in our evaluation that fairly complex reasoning can be performed rather quickly (in a matter of few seconds in the worst case) on realistic queries and on large data. However, the reader should keep in mind that there could be worst-case scenarios (which do not seem to appear on current data) where the performance is significantly worse, and this is mainly due to the theoretical high worst-case complexity that is inherently present in the reasoning process.

5.6 Related Work

Applying rules with a top-down method like backward-chaining is a well-known technique in rule-based languages like Datalog [17]. In this work, we optimized the computation to exploit the characteristics of RDF data and execute a standard set of rules. Our backward-chaining algorithm is inspired by the QSQ algorithm and the traditional semi-naive evaluation algorithm which are well-known techniques in logic programming. A similar termination condition to ours is employed also in the RQA/FQI algorithm [56].

In our approach, we exploit the availability of the precomputation using a *sideway information passing (SIP)* technique during the execution of the rules. This technique is used in other approaches like in the magic set rewriting algorithm [7]. However, while the magic set algorithm uses it at compile-time to construct rules bottom-up, we employ this technique at runtime to execute queries in a top-down manner. Also, SIP strategies are similarly used in generic query processing to prune irrelevant results. In [39] the authors propose two adaptive SIP strategies where information is passed adaptively between operators that are executed in parallel.

Some RDF Stores support various types of inference. 4store [68] applies the RDFS rules with backward-chaining. Virtuoso [23] supports the execution

of few (but not all) OWL rules. BigOWLIM [13] is a RDF store that supports the OWL 2 RL ruleset by performing a full materialization when the data is being loaded. Another database system that performs OWL RL reasoning in a similar way is Oracle: In [44] the authors describe their approach reporting the performance of the inference over up to seven billion triples. Another approach in which the OWL RL rules are used is presented in [72] where the authors have encoded OWL RL reasoning in the context of embedded devices, and therefore optimizing the computation for devices with limited resources.

Some work has been presented to distribute the reasoning process using supercomputers or clusters of machines. In Chapter 2 we used the MapReduce programming model to improve the scalability. In [94], the authors implement RDFS reasoning using the BlueGene supercomputer. To the best of our knowledge none of these approaches supports the OWL RL rules.

Implicit information can be derived not only with rule-based techniques. In [65], the authors focus on ontology based query answering using the OWL 2 QL profile [91] and present a series of techniques based on query rewriting to improve the performance. While we demonstrate inference over a much larger scale, a direct comparison of our technique with this work is difficult since both the language and reasoning techniques are substantially different.

A series of work has been done on reasoning using the OWL EL profile. This language is targeted to domains in which there are ontologies with a very large number of properties and/or classes. [22] presented an extensive survey of the performance of OWL EL reasoners analyzing tasks like classification or consistency checking. Again, the different reasoning tasks and considered language make a direct comparison difficult for our approach.

5.7 Conclusions

Until now, all inference engines that can handle reasonably expressive logics over very large triple stores (in the orders of billion of triples) have deployed full materialization. In this chapter we have broken with this mold, showing that it is indeed possible to do efficient backward-chaining over large and reasonably expressive knowledge bases.

The key to our approach is to precompute a small number of inferences which appear very frequently in the proof-tree. This of course re-introduces some amount of preprocessing, but this computation is measured in terms of minutes, instead of the hours needed for the full closure computation.

By pre-materializing part of the inference upfront instead of during query-time, we are able to introduce a number of optimizations that exploit such

precomputation to improve the performance during query-time. To this end, we adapted a standard backward-chaining algorithm like QSQ to our usecase exploiting the parallelization of current architectures.

Since our approach deviates from standard practice in the field, we have formalized the computation using the theory of deductive databases and extensively analyzed and proved its correctness.

We have implemented our method in a proof-of-concept Java prototype and analyzed the performance over both real and artificial datasets of five and ten billion triples using most of the OWL RL rules. The performance analysis shows that the query response-time for our approach is in the low number of milliseconds in the best cases, and increasing up to few seconds as the query increases in its complexity. The loss of response time is offset by the great gain in not having to perform a very expensive computation of many hours before being able to answer the first query.

Obvious next steps in future work would be to investigate how our approach can further scale in terms of data size and reasoning complexity and to understand the properties of the knowledge base that influence both the cost of the limited forward computation and the size of the inference tree. Also, it is worth to explore whether related techniques such as ad-hoc query-rewriting like the one presented in [65] can be exploited to further improve the performance.

To the best of our knowledge, this is the first time that complex backward-chaining reasoning over realistic OWL knowledge bases of ten billion triples has been realized. Our results show that this approach is feasible, opening the door to reasoning over much more dynamically changing datasets than was possible until now.

Chapter 6

Reasoning and SPARQL on a distributed architecture

In Chapter 5 we described a method to reduce the computation of traditional backward-chaining by precomputing some selected queries. We focused our attention on the correctness of such approach, demonstrating that its application is harmless in a way that no derivation is missed.

In this chapter, we complete our discussion describing the implementation of hybrid reasoning on a distributed architecture. In other words, if before we explained "what" we are doing, now we will discuss "how" we can do it.

To this purpose, first we have grouped the rules in our ruleset into four abstract categories and propose a different rule execution algorithm for each of them. Then, since normally users interact with RDF knowledge bases using the SPARQL language, we also explore the possibility of interleaving the rules execution with the processing of *basic graph patterns* of SPARQL queries [66] and evaluate the performance using a distributed (also called shared-nothing) architecture.

As we will describe in the remaining of this chapter, implementing efficient reasoning (and querying) on a large input is a challenging task and a naive implementation might lead to severe performance problems. We addressed some crucial issues in designing our system architecture exploiting the fact that some queries are already precomputed as described in the previous chapter.

We have implemented our method in the QueryPIE prototype and eval-

uated the performance on artificial data of a size between one and hundred billion triples. The evaluation shows that our prototype is able to execute small but realistic queries on a dataset of a size up to hundred billion triples (which accounts to about three times the estimated size of the entire Semantic Web) using 16 computational nodes with a response time that is often under the second. To the best of our knowledge, such performance sets a new limit of the amount of Semantic Web data that is possible to query using a moderated-size network of machines.

We have organized the remaining of this chapter as follows: Section 6.1 contains a description of the overall system architecture reporting some initial assumptions and challenges. After this, in Section 6.2, we describe how the data is stored and distributed across the nodes. Next, in Section 6.3, we describe in detail the rule execution and in Section 6.4 how this process is interleaved with the execution of the multi-patterns SPARQL queries.

In Section 6.5 we present the evaluation of our method using as example the LUBM SPARQL queries. In section 6.6 we report on related work and in Section 6.7 we indicate directions for future work and draw the conclusions of this work.

6.1 System architecture

We identify two main tasks for our system: (i) perform reasoning by applying a set of rules to derive triples that match the input triple patterns; (ii) join the bindings of the triple patterns in the *BGP* of the SPARQL query and return the results to the user.

From a conceptual point of view, these two operations are independent from each other. In principle, we can abstract the task of applying the rules from the broader context of answering SPARQL queries and define it as a process that receives as input a triple pattern and returns a set of bindings. Even though such independency holds at a conceptual level, in practice these two tasks are often related because of efficiency reasons. For example, it is a common practice to push down unary SPARQL operators to the physical data layer in order to limit the number of retrieved bindings.

Initial Assumptions. We consider computational clusters as our physical architecture of reference. In this context, we are called to partition both data and computation across a set of loosely coupled machines. One of the machines in the network is elected to be the interface between the user and the cluster. This machine is responsible to submit the SPARQL queries for the execution and collect the results of the query. In principle, in our architecture every

machine can accept a query and collect the results. Because of this, we can abstract our physical architecture to a logical network where every node has stored a chunk of the data and can communicate one-to-one with all other nodes.

We exclude from our discussion the challenging and important problem of determining the best query plan and we assume that such plan is known beforehand. Further investigation on this issue should be seen as future work.

Also, we designed our architecture optimizing the computation to efficiently answer queries that do not require the processing of a large amount of data. We argue that applying inference at query time with large analytical queries is in principle an inefficient choice, because the additional computation would become too expensive to perform at query-time. Techniques based on forward-chaining where the entire inference is precomputed have shown to be very efficient in handling such cases (as described in Chapters 2 and 4).

Challenges. In Chapter 1 we outlined three main challenges for implementing an efficient distributed reasoning approach. These are: *large data transfer*, *load balancing*, and *reasoning complexity*. The first challenge warns that an approach designed to perform frequent large data transfers is in principle inefficient because such operation is expensive on modern architectures and should be avoided whenever possible. The second challenge is of fundamental importance in a distributed setting because load balancing problems lead to severe performance inefficiencies as the number of machines increase. The third problem is specifically related to reasoning (in contrast to the other two which are common problems in distributed systems) and relates to its high theoretical computational complexity.

We made some choices in designing our system architecture in order to reduce the effect of these three problems on the overall performance and minimize their impact on the scalability. In the next three sections we will point out how such choices relate to these issues.

6.2 Data Storage

The distribution of the data influences how the computation should be parallelized. For example, the MapReduce programming model exploits the fact that the data is not indexed in order to efficiently parallelize the map tasks. On the contrary, relational database technology very often heavily relies on the data indices to achieve high performance data lookups. In our context, the response time is an important constraint and this makes it essential that we build and maintain a series of data indices.

A complete description of the physical storage of our system model goes beyond the scope of this chapter. Here, we will limit to provide a high level description necessary to understand the overall architecture.

First of all, we make an important distinction between the storage of triples that are part of the precomputed patterns (these triples are materialized before query time as explained in Chapter 5) and the other ones. The precomputed triples (i.e. the ones in the edb relations S_Q in Algorithm 13) are replicated on every node and stored in a series of in-memory hash maps. On the contrary, all the other triples are not replicated, but rather indexed and partitioned across the nodes*.

Currently, maintaining a series of indices with all the possible triple permutations is the current state of the art for RDF data [96]. Our approach follows this lead and creates six different indices with the permutations *spo*, *sop*, *pos*, *pso*, *ops*, and *osp*.

These indices are created and range-partitioned across the nodes of the network using a MapReduce based algorithm. The method that we used relies on two MapReduce jobs: The first job calculates with sampling the statistical information necessary to find the boundaries of the partitions of the index, while the other uses this information to equally partition the triples and perform a global sorting. After the indices are created, we store the data in an on-disk data structure. To this end, we do not use a traditional B+Tree to store the entire triple, but we have rather implemented a variant of B+Tree to retrieve the first element of the index and store the list of the second and third elements in a block-based list.

In our adopted method, the choice of replicating the precomputed triples on all nodes has the beneficial effect of reducing the problem of large data transfers. Since all of them are available locally, our sideways information passing strategy (described in Section 5.2.1 of Chapter 5) can be implemented locally without any data transfer.

Also, the decision of range-partitioning the indices is of crucial importance to understand the load balancing properties of the system. In fact, if the data is range-partitioned then most likely only a subset of the nodes contains the relevant data to answer the query. If the query requires the access of a small amount of data, then such choice is ideal because the overhead of contacting all the nodes will be higher than the advantage of exploiting the total computational power of the cluster. However, in case the query requires the access to a large collection of data, then such solution might introduce

*Note that these indices contain also a copy of the precomputed triples that are also available in the in-memory hash-maps. This is to guarantee the completeness of the algorithm (see Chapter 5 for more details).

load-balancing problems because only the machines that store the data can execute the query (while the others are unused). By range-partitioning the data, we are indirectly optimizing our system to efficiently answering small queries, and such conclusion is inline with our initial assumptions to focus on the performance over small queries.

6.3 Rule Execution

In the previous section we described how the data is stored and distributed in our architecture. In the next two sections we will focus our attention on the distribution of the computation and describe the execution of the inference rules and of the SPARQL queries.

The first stage in the execution of our system consists in performing the prematerialization of some specified triple patterns using the Algorithm 13 described in Chapter 5. This operation is performed before the user can query the data and it is necessary in order to guarantee the completeness. After this algorithm is terminated, all the precomputed triples are replicated on each node and at this point the system is ready to accept user queries.

At this point, once the system has received a SPARQL query in input, it proceeds retrieving the triple patterns that are part of the query and, for each triple pattern in the query, the system invokes backward-chaining reasoning using Algorithm 12.

The backward-chaining reasoning process is described in detail in Chapter 5. In brief, given an input triple pattern, the program constructs a tree (called *proof tree*) which has the original query as root and several subqueries as leaves. The nodes of this tree contain the rules that might derive triples relevant to this query. These rules will receive in input the triples that come from the lower levels of the tree and produce derivation that will serve as the input of the rules at the parent level.

In this chapter, we will not describe the execution of the backward-chaining algorithm because such discussion was presented in the previous chapter. Here, we will rather focus on the execution of the single rules that are applied in this process and describe how we execute them in our distributed architecture.

Physical rules execution. We can abstract the application of a rule as a generic *action* which receives some triples in input and returns some others in output. Consequently, a path from the root of the tree to a specific leave can be seen as a sequence of actions or, as we call it in our system, a *chain* of actions.

Cat.	Body	Head
1	$T(X, SCO, Z)$, $T(Z, SCO, Y)$	$\Rightarrow T(X, SCO, Y)$
2	$T(A, TYPE, X)$, $T(X, SCO, Y)$	$\Rightarrow T(A, TYPE, Y)$
3	$T(P, TYPE, TRANS)$, $T(X, P, Z)$ $T(Z, P, Y)$	$\Rightarrow T(X, P, Y)$
4	$T(C, INTER, X)$, $LIST[X, C_1, ..., C_n]$ $T(Y, TYPE, C_1)$ ... $T(Y, TYPE, C_n)$	$\Rightarrow T(Y, TYPE, C)$

Table 6.1: Example of OWL RL rules in each category. We used the abbreviations reported in Table 5.1 to represent standard URIs. For conciseness we introduced the predicate `LIST` to indicate a list of terms encoded using the RDF syntax.

A chain of actions is the unit of computation in our architecture. When a computing node receives a triple pattern as input, it generates several chains of rules that correspond to the paths of the proof tree for the given query.

These *chains* start with a triple pattern that must be read from the physical data layer. Depending on the triple pattern, the appropriate index is chosen and the chain is sent to the nodes that might contain relevant data. All triples that are read from the data layer will be passed to the first rule in the chain until the last one will return triples to be added in the answer set.

The physical execution of a rule depends on the number and type of its antecedents. We identified four categories into which each rule can be classified:

1. Rules that have as antecedents only precalculated patterns;
2. Rules that have as antecedents one or more precomputed patterns and exactly one generic pattern;
3. Rules that have as antecedents a fixed number of generic patterns;
4. Rules that have as antecedents a variable number of generic patterns.

In the remaining of this section, we will describe how rules of each category are executed in our system. To ease the comprehension, we report in Table 6.1

Algorithm 14 Algorithm for the execution of rules of the 2nd category. This code is executed only once at loading-time. r is the considered rule while *Database* represents the input data (explicit triples and partial derivation)

```
1   precomp_t := ∅ //Contains the join results of the precomputed patterns in the rules body
2   gen_p := ∅ //Generic pattern in the rules body
3   for(∀ p ∈ r.BODY)
4     if p ⊑ PrecompPatterns then
5       tuples := lookup(p, Database)
6       if tuples = ∅ then
7         return ∅ //rule is not active
8       end if
9       precomp_t := precomp_t ⋈ tuples
10    else
11      gen_p := p
12    end if
13  end for
14  shared_vars_h := { var | var ∈ rule.HEAD.vars ∧ var ∈ precomp_t.vars}
15  shared_vars_p := { var | var ∈ gen_p.vars ∧ var ∈ precomp_t.vars}
16  for(∀ v ∈ shared_vars_p)
17    val_v := π_v(precomp_t)
18  θ_g := MGU(inst_head,gen_p)
```

an example rule for each of them. We will start from the simplest category and finish with the most challenging one.

Category 1 In our system, the precomputed triples are replicated on all nodes. Because of this, all the rules in this category can be executed locally, without any communication between the nodes, simply performing a hash join between the in-memory data structures. In case the number of triples to join is large (e.g. above a specified threshold), the join is split in several threads and executed in parallel within the single node. Since these joins are executed locally, the execution of these rules do not relate to the problems of large data transfer and load balancing among the nodes.

Category 2 Rules in this category require a join between a set of triples that is locally available in main memory, (i.e. the precomputed triple patterns) and a generic triple pattern that is read from the input of the chain. Because the triples from the generic pattern might reside on different nodes, the execution of rules in this category might either require some data transfer between the nodes or cause load balancing problems. Therefore, the execution strategy must address these issues properly.

In our explanation we will use the second rule of Table 6.1 as an example, but the algorithm is generic and is applied for every rule in this category.

Algorithm 15 Algorithm for the execution of rules of the 2nd category. This code is executed when the rule is triggered to a answer a given query. In this algorithm r is the considered rule. *Database* represents the input data (explicit triples and partial derivation), and Q is the input query and *inst_head* is the instantiated head.

```
1   if shared_vars_h = ∅ or all shared_vars_h in inst_head are not bound then
2     inst_gen_p := θ_g(inst_head)
3     for(∀ v ∈ shared_vars_p)
4       inst_gen_p.v := val_v // Set a reference to ‘‘v’’ in inst_gen_p
5     retrieved_tuples := infer(inst_gen_p,Q)
6   else
7     filteredPrecomps := inst_head ⋈ precomp_t
8     retrieved_tuples := ∅
9     inst_gen_p := θ_g(inst_head)
10    for(∀ t ∈ filteredPrecomps)
11      θ_p := MGU(filteredPrecomps,inst_gen_p)
12      retrieved_tuples := retrieved_tuples ∪ infer(θ_p(t),Q)
13    end for
14  end if
15
16  all_subst := retrieved_tuples ⋈ precomp_t
17  return ⋃_{θ∈all_subst} θ(Q)
```

We can divide the execution of these rules in two parts. The first part is executed only once during loading time because it does not depend on a specific input query while the second is executed to answer a given query. We reported in Algorithm 14 the operations that are performed at loading time. First, the algorithm identifies the precomputed triple patterns in the body of the rule (line 5) and joins them together using an hash join. This operation is performed locally, without any node communication. In case the join does not produce any tuple (line 7) then the rule is deactivated since it cannot produce any derivation.

After this, the algorithm calculates the variables that are shared between the head of the rules and the precomputed patterns (line 15) and between the precomputed patterns and the generic one (line 16). For each of these variables, the algorithm performs a projection on the precomputed joined tuples to retrieve all their possible values (these are stored in the sets val_v). The value of these variables in the input query is used at query time to determine the execution strategy.

After these operations are completed the rule is ready to be invoked in order to answer a given query and we report the operations that are performed to this purpose in Algorithm 15.

Here, the algorithm receives in input a query Q and needs to join the triples

parts of the triple body in order to return conclusions that match the query Q. These triples can be divided between the ones that belong to the precomputed patterns and the ones that belong to the generic one. While all the triples that are part of the precomputed patterns are already available in main memory, the triples that match the generic patterns must be retrieved from the data layer (or by the recursive application of other rules). Therefore, the first task of the algorithm consists of retrieving such generic triples. This operation is performed in two ways, depending on the value of the variables in the input query:

- In case the shared variables in the input query are unbound (e.g. the input query is *T(X,TYPE,Y)*), then we are unable to restrict the search to some of the possible values in the precomputed set, and we must perform a join between all the precomputed triples and the generic pattern.

  Since all the possible values of the shared variables were precalculated during query-time (in the variable v_*), and are globally accessible in main memory, the rule executor can set as value of the shared variable in the new query a reference to these values (line 4) and invoke the function *infer* to retrieve all the triples (notice that the value of variables in the head which are shared with the generic pattern are passed directly using the function θ_g). In our example, this would mean that the system will launch a query like *T(A,TYPE,ALL_SUBCLASS)* where `ALL_SUBCLASS` consists of a reference to all the terms that appear as subject of the subclass triples.

- In case the shared variable in the head is bound to a value (or a set of values), we are able to restrict the set of possible values to use in the generic pattern by performing a join between the precomputed tuples and the values of the head. Therefore, for each of the resulted values, we generate a new query substituting the shared value of the generic pattern with each of the possible values from the precomputed set.

  In our example, such condition would appear if there would be an input query like *T(X,TYPE,c)* and c is a class with a number of generic subclasses c_i. In this case, the algorithm would retrieve all the subclasses of c and for each of them launch a new query *T(X,TYPE,*c_i*)* (line 12).

In both cases, after the algorithm has retrieved the inferred triples it still needs to perform the join with the precomputed tuples to calculate the actual derivation (line 16). In our case, such operation is translated with an hash

join between the triples that are passed from the input of the chain and the precomputed tuple set that is available in main memory.

The rule execution strategy that we just proposed tackles the problem of avoiding the transfer of large amount of the data because: (i) in the first case we only need to transmit a reference to a set of values available on each node, and, by transmitting only a reference instead of generating as many queries as all the possible values, we reduce the overhead of launching multiple queries; (ii) in the second case we rely on the assumptions that given a specific class the number of subclasses is limited and therefore the number of queries is small.

Also, such execution strategy limits possible problems of load balancing because the join between the generic and the precomputed triples is always executed locally since one side is always available in memory, and this makes the data join an "embarrassingly parallel" operation. Because of this, the execution of these rules does not introduce any load balancing problem between the nodes as it could potentially happen if, for example, the data is incorrectly partitioned.

Categories 3 and 4 Rules of these categories are more challenging than the previous ones because they require a join between a number of generic triple patterns. In case of rules of the third category, the number of generic patterns is predefined. However, if the rules belong to the fourth category, then the problem is worsened by the fact that the number of generic pattern is not predefined and must be dynamically generated from the input data.

As an example, consider the fourth rule in Table 6.1. Such rule allows us to derive that a particular term x is an instance of a generic class C if C is defined as the intersection of n classes and that the term is an instance of all of them. The length of the intersection is variable and it is specified with RDF triples.

In RDF, a list of terms is encoded using the predicates `rdf:first` and `rdf:next`. Therefore, the first problem consists of calculating the elements of the list by retrieving all the "first" and the "rest" RDF triples and reconstructing the content of the lists by joining these triples.

We have observed that all the lists that are used in the rules antecedents share a variable with one precomputed pattern and we empirically measured that the number of such lists is not large. Such observation is to be expected since normally there is a 1:1 relation with the precomputed patterns they are joined with (for example, a generic class C is normally defined with one or at most very few lists of intersections). Therefore, since the number of these lists is limited, we implemented an algorithm that precalculates the content of all

the lists at loading time by launching two queries that retrieve all the "first" and "rest" triples, and replicating the list contents on all the nodes.

After this operation is performed, the main problem in the executing rules of these categories is that they require a join between triples that reside on different nodes. Because of this, while eventual joins between precomputed and generic patterns in the rules body can be performed as described before, in order to execute a join between the generic patterns a data transfer between the nodes cannot be avoided in any case.

In our system, in order to implement this operation we implemented the strategy of broadcasting one side of the join to the other nodes and perform the final join on a single machine using an hash join. As an example, suppose that one rule requires to join two generic patterns g_1 and g_2. First, the algorithm will query the knowledge base with g_1 and collects all the triples on a single node. Then, it will calculate all the possible values of the shared variable with the next generic pattern and broadcast them to all nodes. After this, it sets a reference to this set of values in g_2, and query the knowledge base collecting all the results in the same node where the actual join is computed.

Broadcasting one side of the join to all the other nodes is an expensive operation that should be avoided whenever possible. To this end, we have implemented a caching mechanism to avoid to broadcast a set of values if this was already done before and in our experiments we have observed that such caching mechanism is very effective since it prevented broadcasting most of the times. Given that our initial assumption was to design a distributed architecture to execute small queries we conclude that such execution strategy suffices.

However, if the number of intermediate triples was to be large then such execution strategy could potentially generate a load unbalance between the nodes since all the intermediate results must be collected on a single machine and a caching mechanism (which relies on set comparison) could become too expensive to be perform every time. Since such cases are beyond the scope of our architecture, here we will simply acknowledge their existence and future research is necessary in order to efficiently deal with them.

6.4 SPARQL queries

SPARQL is a broad and generic language, but at its core each SPARQL query corresponds to a graph pattern matching problem. In this chapter we consider only the most popular category of the possible SPARQL graph patterns that are the *basic graph patterns (BGP)*. These patterns are simply defined as a

set of triple patterns. We require that each triple pattern share at least one variable or RDF term with at least another pattern. Therefore, given a BGP G, for all triple patterns $t_1 \in G$ there must be another $t_2 \in G$ so that at least one variable or term in t_1 is also in t_2.

Algorithm 16 Overall algorithm of the execution of a SPARQL query

```
 1  R ← infer(Q[0])
 2  for i = 1 to Q.length
 3    if R = ∅
 4      return ∅
 5    for ∀ V ∈ R.Varset ∩ Q[i].Varset
 6      send_to(all_nodes,R.V)
 7      Q[i].V = {R.V}
 8    P ← infer(Q[i])
 9    R ← P ⋈ R
10  return R
```

Performing efficient SPARQL query answering in a distributed scenario is a well-known challenging task. In our context, the problem is even more complex because we intend to enrich this process by adding just-inferred triples to the query results.

In our system, we implemented a sequential SPARQL execution strategy where the triple patterns that constitute the SPARQL query are retrieved and joined together one by one. At each step, the intermediate results of the SPARQL query are "pushed down" in the reasoning process until the join is executed directly in the data layer.

To illustrate this methodology in more detail, we report in Algorithm 16 the overall algorithm. The input consists of the sequence of triple patterns Q that constitutes the body of the SPARQL query. At first the algorithm retrieves all the bindings of the first triple pattern (line 1). In case the query consists only of one pattern, the result is immediately returned.

Before proceeding joining the other patterns in the query, the algorithm broadcasts the bindings that will be used in the next join to all the nodes (lines 5 and 6). After this, the algorithm substitutes the original variables name in the next pattern in the list with a reference to the bindings calculated (line 7).

The operation of broadcasting a set of intermediate bindings and set a reference in the following queries is similarly performed when the system executes rules with multiple generic patterns. However, in this case such operation does not replace the actual join between the patterns in the SPARQL query. It simply reduces the reasoning process to infer only triples which are relevant to the SPARQL query.

Therefore, once the algorithm has retrieved the other pattern, it still has to perform a join between the two patterns (line 9). This join will have a hit ratio of one or more, since the terms were already filtered when being retrieved from the knowledge base.

The advantage of integrating SPARQL query execution and the inference process in this way is three fold: (i) the rules execution process becomes independent from the SPARQL query execution because the list of bindings is treated as a single entity, as it is during the execution of rules of the third category; (ii) we avoid an exponential generation of chains by substituting each single value of the bindings in the new pattern. Therefore, by pushing down the set of values instead of generating a single branch for each possible combination we reduce the overhead of performing multiple lookups in the knowledge base; (iii) if we perform the joins directly on the data layer, then we can exploit the fact that the index is sorted, and implement this operation using a fast merge join.

It is quite frequent that SPARQL queries define a list of patterns that share only one variable and have only RDF terms as remaining elements. Such queries provide for an interesting optimization to reduce the computation.

For example, suppose we have the following list of patterns: `?p :worksFor :University` and `?p :type :Person`. Here, a generic SPARQL engine will first retrieve the pattern `?p :worksFor :University` and then move to the second one.

In our case, after the engine has retrieved all the results for the first query, it will set a reference to the values in the second pattern and proceed querying the knowledge base. However, instead of invoking the complete reasoning algorithm a second time, it will simply retrieve the explicit triples and apply only once rules of the first and second category. These rules are not executed in parallel but rather in a sequence (in our architecture we can set how many of these rules can be executed at the same time). Whenever triples are inferred using this simplified version of reasoning, the algorithm removes the corresponding binding from the reference set of values used in the second pattern (i.e. all the "?p" calculated with the first pattern). If at the end of this process there are still some "?p" which are not verified, then the algorithm will repeat the query activating this time complete reasoning. Otherwise, since all the bindings were already verified, it proceeds evaluating the following pattern (if any).

This optimization relies on the assumption that most inference can be calculated just with a single application of the inference rules. For this particular type of queries, such optimistic strategy can greatly reduce the computation of reasoning avoiding to evaluate the complete proof-tree.

The disadvantage is that such optimization can be applied only if the patterns share a single variable. If the second pattern contains an additional variable that is not part of the join, then such optimization cannot be applied because then we are required to perform complete reasoning in any case to derive all the values of the additional variable.

6.5 Evaluation

We have implemented the algorithms described in this chapter in the QueryPIE prototype. For the communication between the nodes, we relied on a set of Java libraries called Ibis [6] that allows the abstraction of the physical nodes as a logical network.

The LUBM [30] benchmark tool is the *de facto* standard for measuring the performance of OWL reasoners over large knowledge bases. It gives the possibility to generate an input with a variable number of triples using a simple ontology for the university domain. Using this tool we generated three synthetic datasets of 1, 10, and 100 billion triples.

As part of the benchmark there is a set of 14 queries that can be used to query the data. These queries are of various types: some are selective and return a limited number of results, while others require the processing of a large amount of data and return many billion results. The proposed architecture is designed to answer queries of the first type; therefore we exclude the last type of queries (4 out of 14) and analyze the execution of queries that do not return more than a few million results. For convenience, we reported the list of the considered queries in Appendix B.3.

The evaluation was performed on the DAS-4 cluster[†], which is a six-cluster wide-area distributed system. Each cluster is interconnected with a wide-area connection based on light paths of 10 Gbit/s. Each cluster consists of a number of machines with heterogeneous hardware. In this evaluation, we used the cluster at our university that consists of 74 nodes each equipped with a dual quad-core CPU of 2.4GHz, 24 GB of main memory and an internal storage of two disks of 1 TB in RAID-0 mode. For this experiments we chose to use a low-latency InfiniBand connection of 40 Gbit/s instead of the standard Gigabit Ethernet.

To the best of our knowledge, currently there is no other work described in the literature where a complex ruleset like the one considered in this chapter is applied on a very large input. Current approaches either calculate the entire

[†] `http://www.cs.vu.nl/das4/`

Q.	Resp. time (ms.)		Results	I/O access		Cache	
	Cold	Warm	(#)	# lookups	GB	Hits	Miss
1	364.05	20.28	4	17	0.02	0	0
2	368.19	25.12	6	73	0.03	0	0
3	2041.20	493.89	34	19218	0.19	87	3
4	1283.88	82.42	719	194	0.5	0	0
5	2061.27	361.11	4	15493	0.29	100	2
6	11536.95	2784.72	7790	22220	5.90	200	6
7	1745.20	367.96	4	15461	0.09	100	2
8	1664.57	74.03	224	116	1.11	1	3
9	5717.32	1163.61	15	17580	2.75	118	4
10	12304.29	886.68	37118	188	15.01	0	0

Table 6.2: Response time of the LUBM queries on 1B triples

inference before query time or cannot scale to our scenario. Because of this, we are unable to compare our performance with a baseline method.

Therefore, we designed a set of experiments in order to understand the strong and weak points of our approach. We grouped them in three sections, each with a different goal: In Section 6.5.1, we aim to measure the absolute performance of our system. Next, in Section 6.5.2 we analyze the scalability of our approach. Finally, in Section 6.5.3 our purpose is to evaluate the overall efficiency of our system by analyzing the performance of the most expensive query.

6.5.1 Performance

For this set of experiments we use as input a LUBM dataset of about 1 billion triples, and distribute the computation over 8 DAS-4 nodes.

The first operation (after the data compression) consists of creating the six data indices using MapReduce. To this purpose, we have written a program using the Hadoop framework and deployed a small Hadoop cluster on 8 nodes of the DAS-4 cluster. The operation of generating the indices took about 48 minutes. The execution time increases linearly with the input size.

After we generated the indices, we calculated the precomputed patterns using the method explained in Chapter 5. This method consists of repeating the execution of a set of queries until no further derivation is returned.

After the process of data preparation is finished, the users are able to query the datasets. In Table 6.2 we report the execution of the LUBM queries on

the same dataset activating the execution of the OWL rules at query time.

The second and third columns report the response time of the system. The time is measured from the moment that the query is launched to the time where all the data is collected at the interface node. The second column reports the cold runtime, that is the first runtime after the system is booted and all data needs to be read from disk. The third column reports the warm runtime, that is an average of the 29 consecutive runs of the same query. These last runs are performed where all the data is in memory and no disk access is performed.

The fourth column of the table reports the number of results. The fifth column reports the number of data lookups performed on the data layer that were necessary to apply the rules and execute the query. This number can be used as an indicator of the complexity of the query. For example, both queries 1 and query 2 trigger a relatively low number of data lookups because not many rules are executed. In contrast, the execution of query 6 requires more than 20 thousands data lookups, and this indicates that the query is much more complex than the previous two. Another indicator of the cost required to execute the query is given in the sixth column that reports the number of bytes read from the data indices during the first execution of the query. As we can see from the table, some queries require a considerable amount of data to be read (the worst case is for the last query where about 15 GB of data are read from disk).

The last two columns of the table contain the number of cache hits and misses during the execution of rules of the third and fourth category. As we can see from the results, most of the times the cache is able to avoid a data broadcast of the data since the number of hits is far higher than the number of misses.

From the results presented in the table, we notice that the warm runtime is always considerably lower than the cold runtime. This is due to the fact that during the first run most of the data must be read from disk. On average the cold runtime is in the order of few seconds while the warm runtime is almost always below the second, which we considered to be an interactive response time. The slowest warm runtime is reported for query 6 where it takes about two seconds. The slowest cold runtime is of query 10 where it takes about 12 seconds to read and process 15GB of data. In our experiments this last query revealed to be the most expensive as we increased the data size since it produces a large amount of results and requires a fairly complex rules execution. In Section 6.5.3 we take it as a test case and analyze its execution in detail.

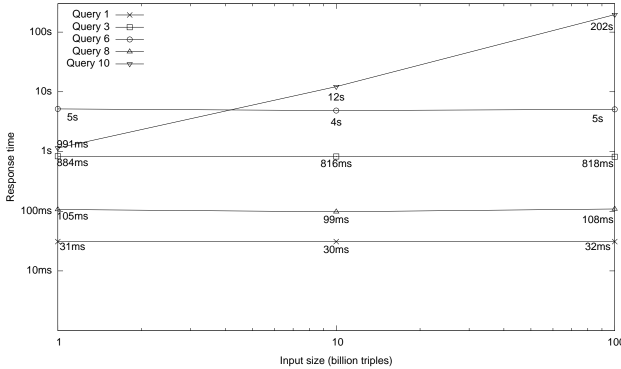


Figure 6.1: Response time of LUBM queries varying the input size

6.5.2 Scalability

At first, we evaluated the scalability of our architecture by increasing the input size and maintaining the number of nodes constant.

To this purpose, we have generated LUBM datasets of 1, 10 and 100 billion triples and launched the LUBM queries using 16 nodes. In Figure 6.1 we have plotted the response time of five LUBM queries against the input size. From the figure, we notice that in four cases the response time remains constant. This happens for query 1, 3, 6 and 8. The explanation for such performance is that these queries consider an amount of data that is constant regardless of the total input size. Therefore, our architecture is able to efficiently retrieve maintaining the response time fairly constant even the input increases.

The performance of query 10 exhibits a different behavior: Here, the data considered is not constant but increases with the input size. The number of results changes with the data size as well: with an input of one billion triples the output is about 37 thousand triples, while with a hundred billion it is about 3.7 million. Because of this, the workload increases proportionally and so does the total response time: while for one billion it was about a second, with a hundred billion it becomes about 195 seconds. The linear correlation between the input and the response time indicates that the performance of our architecture for this query does not degrade as the input size increases.

In another set of experiments we kept the input size constant and changed the number of nodes in the computation. We used the dataset of 10 billion triples and executed three example queries doubling the number of nodes starting from one up to 32. In Figure 6.2 we report the response time of both the

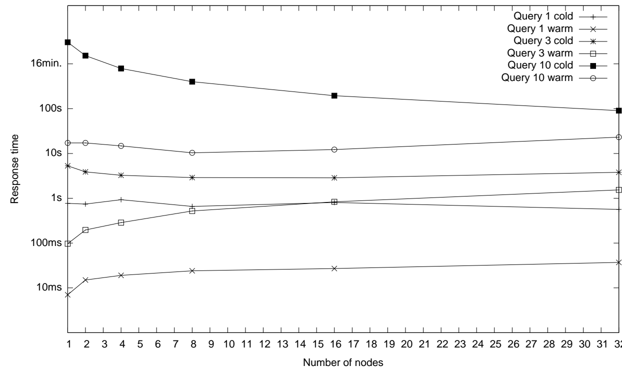


Figure 6.2: Response time of LUBM queries varying the number of nodes

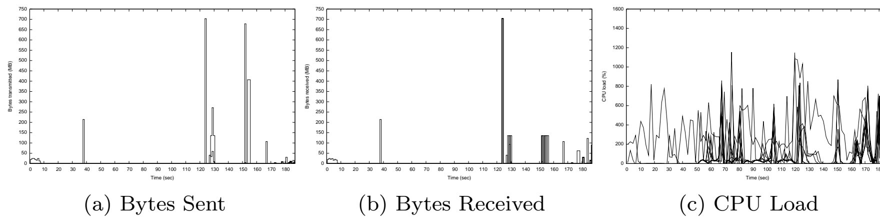
cold and warm runtime. Also in this case the performance of the queries is not uniform.

In general, we observe that the cold execution time of the queries decreases significantly as we increase the number of nodes. For example, the execution of query 10 using one node takes about 50 minutes while it takes around 90 seconds using 32 nodes. The reason for this is that at the first run all the information must be read from disk and if the data is distributed on more nodes then this process is parallelized on more disks with a higher total bandwidth.

The same does not hold for the warm execution time of the queries. In this case all the data is already loaded in memory and the disk access is no longer the performance bottleneck. The execution time becomes in the order of hundreds of milliseconds which is too small to benefit from the distribution of the computation. In this case the performance bottleneck is represented by the synchronization process between the nodes. Since the number of nodes increases, the total response time is affected negatively by it and we observe a decrease of performance as we increase the number of nodes.

We also observe a little fluctuation in the performance of the single runs that is most likely due to the Java garbage collector which sometimes is being activated during the computation. To improve this issue, we have used different garbage collectors algorithms and noticed that indeed such aspect can significantly influence the performance.

Summarizing, we conclude that in our experiments our architecture is able to maintain the response time constant for queries that involve a fixed amount of data regardless of the input size. If the query workload increases with the



(a) Bytes Sent (b) Bytes Received (c) CPU Load

Figure 6.3: Plot of the network traffic and CPU load during the execution of query 10 over 100 billion triples

input, like for query 10, then the performance exhibits a linear progression.

If we increase the number of nodes, queries with a relatively small workload are penalized due to the increased synchronization cost. In contrast, queries with a frequent disk access or that require a significant workload benefit from having a higher disk bandwidth and the additional available computational power.

6.5.3 Efficiency

In order to understand the performance of the architecture we monitored the execution of the query 10 on the input of 100 billion triples using 16 nodes.

The first execution of the query is dominated by the time necessary to read the data from local disks. Such run takes about 1 hour and about 1.7 TB of data are read from the indices.

The next runs of the query benefit from having the data loaded in memory and during the execution there is no access to disk. In this case the execution time drops to around 187 seconds.

We monitored the nodes during the execution recording the number of bytes sent and received and the CPU load of each node. We present these measurements in Figure 6.3a, 6.3b, and 6.3c. In these graphs, the horizontal axis represents the time in seconds while the vertical axis reports the metric that we measure. Each line represents the behavior of one single node.

We notice that for the first 50 seconds there are only two nodes that are active: their CPU load increases up to 800% even this happens for not more than a second, which means that all the 8 cores of the machine are used (each core has hyperthreading, and that brings the maximum load to 1600%).

After around 50 seconds the CPU activity become more intense and we observe that several nodes become active, even the activity is rather irregularly

spread across the machines. However, we notice that during the computation it never happens that all the cores are busy. Only in very few moments the load reaches the 1200% which indicates that 12 out of the 16 virtual cores are being utilized. Such results lead us to the conclusion that the computation does not utilize all the available cores. Therefore, we expect that the performance will not significantly increase if we increase the number of cores per machine.

If we look at the number of bytes exchanged between the nodes, we notice very little network activity. Only in few moments there is a significant increase in the network workload. However, such increase is not persistent and ends immediately after one or two seconds. Such peaks in the network usage happen when the system is broadcasting the intermediate results to the other nodes. For example, we can observe one of these moments at around 124 seconds where one node broadcasts about 700 MB of data to all the other nodes.

Even in such moments the network becomes the performance bottleneck, we conclude that the network is not playing a crucial role in determining the performance because it is involved only for few seconds over the entire run.

Therefore, by analyzing these results we have observed that there are two main reasons that are limiting the performance of the system. The first consists of the synchronization points, which are used in the execution of rules of the third and fourth category. These points force the system to stop and wait until all the other nodes have finished their computation.

The second reason is that one chain is executed by a single thread and sometimes there are not enough chains to fill the entire computational power of the node, so that the computation is bound by the performance of a single core. In future work we are planning to investigate on these issues in more detail and research for solutions to further increase the performance.

6.6 Related Work

RDF Storage. Our choice to store the six permutations of the triples is inspired by [96]. In this work, the authors use a sequence of in-memory shared and sorted vectors to navigate through the indices. Our approach differs from it because we use different data structures which use a variant of B+Tree to store the first element and sorted block-based lists for the second and third elements.

Another efficient in-memory-only data structure is proposed in [5] where RDF triples are stored in a 3-D adjacency matrix. Since such data structure is not appropriate for sparse graphs, the authors use a gap compression scheme to reduce the memory footprint. While such data structure is ideal for low-

selectivity joins, its usage is limited by the main memory size.

In [61] the authors propose to use a hybrid data structure to store RDF data on disk. Such data structure is a combination of an hash-table and B+Tree to improve the $O(logn)$ worst case of B+Trees with the constant lookup time of hash tables. In order to guarantee constant lookup time, they generate one index per all the 15 possible combination of the triple elements, which become prohibitively expensive for very large collections of data.

A radically different approach is to store RDF data using vertical partitioning, which was initially proposed in [1]. The weak points of such approach is that the performance is not optimal when the predicate is unbound. A comparison of the performance of this approach against conventional tuple-based approaches is presented in [71, 73]. In [37] the authors propose yet another data partitioning criteria where the vertexes of the RDF graph are partitioned using graph partitioning algorithms and the triples are assigned to these partitions accordingly. Then, the triples are assigned to all partitions that contain vertexes that can be reached starting from the triple with a maximum of n hops. If n equals one, this partitioning scheme corresponds to join all triples that share one variable and partition the results across the nodes.

An efficient RDF store is RDF-3X [59] but it does not support inference. 4store [68] applies the RDFS rules with backward-chaining. Virtuoso [23] supports the execution of few (but not all) OWL rules. BigOWLIM [13] is the only database that supports the OWL 2 RL ruleset but it applies the rules in a forward-chaining way when the data is loaded and not at query time.

Implicit information can be derived not only with rule-based techniques. In [65], the authors focus on ontology based query answering using the OWL 2 QL profile [91] and present a series of techniques based on query rewriting to improve the performance. While we demonstrate inference over a much larger scale, a direct comparison of our technique with this work is difficult since both the language and reasoning techniques are substantially different.

In our approach, we use a *sideway information passing (SIP)* technique for the execution of the most complex rules. This technique is similarly used in the magic set rewriting algorithm [7]. However, while the magic sets algorithm uses it at compile-time to construct rules bottom-up, we employ this technique at runtime to execute queries in a top-down manner. SIP strategies are also used for generic query processing to prune irrelevant results. In [39] the authors propose two adaptive SIP strategies where information is passed adaptively between operators that are executed in parallel.

SPARQL. In [58] the authors specifically address the execution of SPARQL queries and present another light-weight method to provide highly effective filters on the query input streams. While these techniques are very effective, it

becomes complex to implement them in a completely distributed setting due to the high level of required synchronization.

In [37] the partitioning scheme allows that some SPARQL queries can be executed efficiently without any communication between the nodes. The queries that still require a communication between the nodes are executed with MapReduce.

In [46] the authors propose three strategies to execute SPARQL queries over a number of sources that are available on the Web. These three strategies require a different minimum level of knowledge about the data sources.

A similar assumption about the distribution of data is done in [32] where the authors present a methodology to answer SPARQL queries using remote data sources. This methodology alternates the execution of the query with the deference of the URIs retrieved so far to discover new information that can contribute to answering the query. These last two works differ from ours because we assume that all the data is locally available so that it does not need to be remotely retrieved during query time.

6.7 Future Work and Conclusions

Summary. The problem that we address in this chapter is to implement the hybrid reasoning algorithms that we presented in Chapter 5 in a shared-nothing architecture and integrate this process within the broader task of answering SPARQL queries.

Notice that the use case that we tackled in this chapter is different than the one considered in Chapter 4. In fact, in Chapter 4 we have considered the execution of very large analytical queries typical of ETL scenarios assuming that reasoning was performed beforehand (for example using the technique described in Chapter 2). These queries require an high computation and this justifies the usage of a programming model like MapReduce. On the contrary, in the current and previous chapters we have focused on the execution of much more specific queries and studied the impact of reasoning directly at query time.

To this end, we evaluated the performance on a distributed scenario using the OWL RL ruleset and the LUBM queries on datasets with sizes between 1 and 100 billion triples. The results show that in general the response time is often under the second which is in the range that we consider as interactive response time. We have also analyzed the scalability by increasing the input size and the number of nodes and analyzed in detail the performance of the most complex query in order to understand what are the factors that are

limiting the performance.

Limitations and future work. From the results that we obtained, we can identify some limitations that appear especially when we increase the input size to hundred billion triples. While our architecture shows good performance when answering small queries, we notice that in some cases the reasoning process produces thousands of data lookups that becomes too expensive to perform due to the extremely large amount of data that needs to be read from disk. A similar situation is reached when the number of lookups is not so high, but the queries are such that they require access to multiple locations in the index. In both cases, the computation becomes too expensive to guarantee interactive response time (and sometimes even to be performed at all).

Because of this, the presented evaluation hints to some directions for future work. One possible research line could investigate whether a different rules application strategy could be more efficient for very large queries. Our optimistic strategy proved to be effective, and a similar principle could be extended to handle other cases.

Further work could investigate whether a data distribution scheme that is different from standard range-partitioning could be more effective to answer larger queries. Also, further research is necessary to implement an efficient query planning strategy that can be used when the cardinality of the triple patterns cannot be estimated directly because the inference could produce an undefined number of triples.

Another limitation is that in this chapter we consider only the execution of basic graph patterns, while the SPARQL language allows other operations like aggregation or projection. Future work should aim to implement the missing functionalities.

One aspect that is not considered in our evaluation are the actual preferences of a potential user of our system (such aspect is also not considered in the evaluation presented in the previous chapters). These preferences could play a very important role in assessing the quality of our approach. For example, if there is a specific use case where a user is interested in only a specific query which is frequently requested, then a more sophisticated caching mechanism could greatly improve the performance. Because of this, a line of future research should aim to apply our method to some specific scenarios, and adapt our system to their specific requirements to achieve better performance.

Finally, from a more high performance computation perspective, it is also interesting to investigate in more detail in which measure the several components of the architecture are responsible for the performance. Such analysis would aim to identify exactly which hardware architecture would be ideal for our approach. To this end, a number of experiments could be conducted on

different computer architectures than computational clusters. For example, it would be interesting to evaluate the performance on SPARC T4/T5 based systems or on others which have a significant larger amount of shared memory (in the orders of terabytes). In this way, we could identify more precisely the bottlenecks of our system and consequently further improve the execution improving the performance by using better hardware.

Conclusions. To the best of our knowledge, our approach is the first to demonstrate not-trivial inference during the execution of SPARQL queries on an input size of 10^{11} billion triples, which is one order of magnitude larger than the current state of the art. These results are obtained by partitioning both the data and the computation on a set of loosely coupled machines. We also show how this process performs on a wide area network simulating a heterogeneous environment with different machines and connections.

This work is the first in its kind and demonstrates that SPARQL queries can be enriched with relevant data calculated with reasoning. While our current implementation shows some limitations in processing large and complex queries due to the very large amount of data to process, we demonstrated that reasoning can be employed in enriching small but yet realistic queries on a truly web-scale. Remaining challenges are to extend it to implement missing functionalities and to further improve the performance to allow the execution of more complex queries.

Part III

Discussion and conclusions

Chapter 7

Conclusions: Towards a reasonable Web

The problem of web-scale reasoning is a crucial obstacle in transforming the original vision of a "Semantic Web" into reality. This problem can be tackled at different levels that range from a theoretical analysis to its physical implementation. At each of these levels, web-scale reasoning poses several research questions of a fundamental importance. For example, from a theoretical perspective it is important to research algorithms with a computational complexity that can be afforded on a large scale with current technologies. On a lower and more technical level, engineering research questions that are concerned, for example, on efficient storage and retrieval of RDF data, become more relevant and have a high-priority.

In this thesis, we chose to approach this problem on the lowest of such levels by researching methods to efficiently implement standard reasoning techniques like forward and backward-chaining on a distributed system. While our evaluation shows that web-scale reasoning is indeed possible, in our research we addressed only a part of the general problem and there are still several open questions to be answered.

To this end, in order to truly solve the problem of web-scale reasoning, extensive research must be conducted at every level in order to provide a general and elegant solution to each of its issues. In this context, we argue that research should not be conducted only "intra-level", but also "inter-level"

because findings discovered at one particular level might become potentially useful to drive research at the other ones. For example, high-level research (e.g. a theoretical analysis or a standardization process) might benefit from being aware of what works and what does not when reasoning is implemented on a distributed setting.

In order to facilitate such "inter-level" research, we believe that it is important to abstract the outcome of the specific work in terms of more generic principles so that they can be used at different levels. For example, on a theoretical level our technical contribution might have little value as it is, since our algorithms do not change the general computational complexity required by reasoning. Nevertheless, with our techniques we have demonstrated reasoning over a very large scale, and the underlying principles that have enabled this (if any) might give useful insight to provide for more suitable theoretical models.

Because of this, in this concluding chapter we take a distance from our technical contribution and analyze it from an higher perspective, with the intention of abstracting it into some high-level considerations that could potentially serve to this purpose. As an additional motivation, we argue that such abstraction process is useful since it could contribute to the definition of a sort of guideline to address similar problems in a distributed system.

For the purpose of this chapter, it is important to choose the right level of abstraction because if we abstract too much then our conclusions might become either trivial or too vague. In contrast, if we do not abstract enough then such conclusions might loose their potential of being useful in a different context.

Given such premises, we decided to reformulate a number of practical considerations that arose from our experience as three "laws" that we believe hold in our context and more arguably even in a more general setting. We must warn the reader that there is a degree of speculation in our claims since such laws are mainly based on empirical findings rather than on universal truths. Therefore, they should be seen more as a lesson learned from experience rather than a number of absolute laws that hold in all their generality.

We titled these three laws as follows:

- **1st Law:** Treat schema triples differently;
- **2nd Law:** Data skew dominates the data distribution;
- **3rd Law:** Certain problems only appear at a very large scale.

We can see each of these laws as a pillar that represents an underlying principle or assumption that was crucial in shaping the performance of our

work. By symbolically standing on these three pillars, the methods that we presented in the previous chapters were able to limit the problems outlined in Chapter 1 and demonstrate reasoning on up to three times the entire size of the Semantic Web. Each of them will be discussed in the following sections. After this, in Section 7.4 we will report the overall conclusions of this thesis.

7.1 1st Law: Treat schema triples differently

In this thesis, we make a fundamental difference during the reasoning process in handling the schema triples (which are extracted from the knowledge base and treated differently) and the other ones. With schema triples we generically refer to those triples that define the domain of the knowledge. Such distinction is very similar to the T-Box and A-Box distinction in description logic. For example, the triple `(:Student :subclassOf :Person)` is a schema triple because it simply states that all students are also persons.

In the reasoning methods presented in Chapters 2, 5, and 6, the schema triples are always replicated on each node to minimize the data transfer during the computation. In contrast, generic triples are not being replicated but rather partitioned according to some criteria. In those chapters we have outlined why such division is beneficial for the computation: it reduces the data transfer and allows the implementation of smart techniques to speed up the execution. In doing so, we heavily rely on the assumption that on the Web there are not so many schema triples while there are many more generic ones. However, although the number of these triples is small, they are fundamentally important because they are frequently used in inference rules as they appear in almost all of the rules antecedents.

Sometimes, determining the set of all the schema triples might be a challenging operation. This is not the case of Chapter 2, because in there schema triples can only be derived after the execution of MapReduce jobs which stores the output on files. Since the data resides on a distributed file system, it is simple to retrieve them in the following reasoning steps. However, in the context of Chapter 5 the situation is not so simple, because schema triples can be inferred at a later stage. This required us to introduce a procedure at the beginning of the computation (and provide for a theoretical ground to verify its correctness) to ensure that this operation is performed without loosing derivations.

Anyway, regardless of how challenging it is to retrieve the set of schema triples, in both cases the evaluation clearly shows that such strategy is beneficial in order to increase the performance. Therefore, with our first law we

state that reasoning methods should aim to consider such distinction because several optimizations can be introduced to make the computation more efficient and the method more scalable. Empirical observations allow us to claim that the computation to make this distinction (i.e. retrieve the schema triples) is not an expensive operation and the gain obtained later certainly justifies the cost of doing it.

7.2 2$^{\text{nd}}$ Law: Data skew dominates the data distribution

In the previous section we stressed the importance of the schema triples and reported on our strategy of replicating them on all the nodes, with the assumption that the number of these triples is small on realistic data.

In this section we move our focus to the other type of triples, which are the generic ones. Because in a large collection of data there can be many of them, a strategy based on replication is not acceptable because the local space on single machines is often not enough. Therefore, we are obliged to partition the input across several machines.

The way data is partitioned across the nodes has a fundamental importance in determining the overall performance of the system. Unfortunately, there is no universal partitioning criteria that fits all needs. In fact, the chosen partitioning criteria determines how and, (even more important) where the computation occurs.

For example, in MapReduce the data is stored on files that reside on the distributed filesystem HDFS where raw data blocks are replicated on a limited number of nodes (three by default). When the Hadoop scheduler receives the request of executing a map task, it schedules it on one of the nodes which has the input locally stored. In this way, Hadoop moves the computation rather than the data and this limits the problem of large data transfer. This is an example of how the partitioning of the data influences the computation because if the data would not be equally split among the nodes, then the scheduler would not be able to provide for an effective way to parallelize this operation.

We have observed in our experiments that current Web data is highly skewed and this makes it more difficult to choose a partitioning criteria to impose a fair balancing of the computation between the machines. Therefore, the impact of data skewness is of great importance on the overall performance and should not be underestimated.

We have observed that data skewness does not always occur. From our empirical analysis, we noticed that skewness appears more frequently on the distribution of the predicates and the objects rather than on the subjects. This means that while there is not a large variance in the number of triples that share the same subject, there is a much larger difference in the number of triples that share the same predicate or object (think for example all the `rdf:type` triples). Such consideration can drive an implementation where, given a generic input query, data is accessed through the subjects rather than the predicates or objects.

In our work, we tackled the problem of data skewness in several ways. In Chapter 4 we have introduced a new join mechanism that performs bifocal sampling to determine which terms can introduce a load balancing problem. A similar mechanism is used also for data compression in Chapter 3 and for the reasoning over the *owl:sameAs* statements in Chapter 2. Furthermore, still in chapter 2, we have further reduced the effect of data skewness by partitioning the data using more than one resource, to avoid cases where the data is grouped by a single popular resource.

In the second part of this thesis, data skewness becomes less important because we focused on the performance of queries which do not require the processing of large amounts of data (and therefore, its impact on the total runtime is minimal). However, if we enlarge the scope of our research to large and complex queries, then data skewness would become an issue and further research would be necessary to achieve high-performance and scalability.

Summarizing, with this second law we intend to warn for the effect of data skewness on reasoning and strongly encourage that a careful analysis is performed on this issue in order to determine whether a generic algorithm can be efficiently parallelized. In our work, we have addressed this issue in several ways: among them, the most general consists of determining with statistical analysis which terms might cause such phenomena and treat them differently. In doing that, we have observed that data skewness is mainly present on the distribution of the predicates and objects and this information can be used to choose, whenever possible, to partition the data according to this criteria (or alternatively using more resources).

7.3 3rd Law: Certain problems only appear at a very large scale

In the context of this thesis, a concrete evaluation of the performance is crucial to assess how our methods would perform in a real scenario. In fact, while a theoretical analysis provides for a good metric to establish the properties of a method, modern computer architectures are very complex systems and only a physical implementation can determine whether one approach indeed "works".

Building a prototype that is able to deal with a massive amount of data is not an easy task. In developing the prototypes used for our experiments we had to address many technical issues and we noticed how particularly important the implementation of a specific algorithm is in order to evaluate its scalability and real performance.

From a research point of view, we are tempted to consider such issues as unimportant because they often do not introduce any theoretical research question. As a result, researchers frequently implement simple prototypes and use them to verify the quality of their contribution. We argue that on a web-scale such approach would be a dangerous mistake, because certain problems appear only on a very large scale (e.g. data skewness) and if the prototype is unable to scale to this extent, then we are unable to verify what is indeed its real performance.

Because of this, we argue that simple proof-of-concepts often do not implement all the necessary elements to be representative. To support our claim, we report some considerations about the development of our prototypes, WebPIE and QueryPIE, highlighting why some purely engineering issues played a fundamental role in achieving high performance. Such issues are not typical of our specific case but are commonly known in the domain of high-performance applications. Therefore, a discussion on them is certainly relevant and potentially useful for similar problems in our domain.

First of all, both WebPIE and QueryPIE are written in Java and consist of respectively about 20000 and 27000 lines of code. The amount of lines can already give an impression of the complexity of the prototypes. The choice of which programming language is more suitable for high-performance applications is a very popular and fairly controversial topic. High-level scripting languages are often used for prototyping because they allow a quick development phase. However, we believe that such languages do not qualify in our context because the process of interpreting code affects too much the overall performance and this makes them not optimal in contexts where every single computing resource is needed.

In our case, we chose Java because all the supporting frameworks and libraries (namely Hadoop and Ibis) were written with this language. Java is among the most popular languages (if not the most popular) and we considered it to be an acceptable compromise since it balances the (little) overhead introduced by the JVM by providing a complete development environment.

The main limitation of using Java for our purpose consisted of the automatic garbage collector, which is an excellent feature but with the limitation that it cannot be controlled by the programmer. Because of this, in our prototypes we developed algorithms that avoided to create new objects but rather reuse existing ones to reduce the impact of the garbage collection phase. We believe that such approach was fundamental in order to achieve the performance reported in this thesis and should be kept in mind for similar problems.

Also, since web-scale reasoning is essentially a data-intensive problem, an efficient memory management strategy is necessary in order to exploit all the resources of the hardware. In our case, we realized that sometimes the standard Java data structures are not appropriate because they are designed to be efficient only if the data is limited and become prohibitively expensive if the input is too large. For example, we experienced that already storing few millions of objects in a map was enough to fill several gigabytes of space in main memory with the standard Java HashMap. To solve this problem, we had to implement custom hash maps which were less memory expensive and rely on data marshaling on byte buffers to avoid the creation of new objects.

Another aspect that cannot be ignored is the storage of the data on disk. In fact, an application that stores the data only in main memory is limited by the size of the machine's memory and this impacts its scalability. Because of this, we had to store the data on disk and this required us to implement mechanisms to quickly retrieve the data from it. To this purpose, we used fast compression algorithms like LZO or Google Snappy to compress the data, instead of more efficient algorithms like Bzip2 which are computationally expensive.

Finally, tuning the configuration of the execution environment has also proven to be crucial in the evaluation. For example, the number of Hadoop mappers and reducers per node or the activation of the data compression for the intermediate results can radically change the performance of WebPIE. In QueryPIE, the size of the internal buffers is a very important parameter, because larger buffers are more difficult to allocate (and might require a call to the garbage collector), while smaller ones get filled more easily.

All these considerations are only examples of the technical problems that we had to solve. All of them do not have a real scientific value because they are commonly known and addressed in existing high-performance applications. Nevertheless, they are necessary components in the scientific process and this

makes the problem of web-scale reasoning very sensible to software engineering issues.

We would like to point out that this has substantial consequences for the evaluation. For example, it is impossible to perform a formal verification of a complex implementation and the correctness can be measured only with an empirical analysis. Also, more complex implementations have higher chances to contain bugs (current industry-delivered code contains on average 10-50 bugs per 1000 lines of code*).

Yet, this task is necessary, especially in our context because too many external factors can influence the performance. Because of this, with our third and final law we intend to redefine reasoning not only as a pure research question but also as an important engineering problem. In doing that, we aim to elevate the engineering efforts that are necessary to implement web-scale reasoning as an important contribution to solve this problem and warn that a wrong implementation can radically change the outcome of the evaluation and therefore of the proposed method.

7.4 Conclusions

The research question that drove the research presented in this thesis was: "How can we perform reasoning to enrich query results over a very large amount of data (i.e. on a web-scale) using a parallel and distributed system?"

Such research question was motivated by the fact that a distributed approach might be able to provide for the computational power necessary to perform reasoning over a very large amount of data with a scalability that could deal with the exponential growth of RDF data on the Web.

Summary. In this thesis, we answered this research question by presenting a number of methods that tackles specific issues of this problem. In the first part of this thesis, we proposed a technique to implement forward-chaining reasoning using MapReduce. Then, we described a technique to improve the performance of reasoning by compressing the input data using dictionary encoding. In the fourth chapter, we proposed a technique that uses Pig to query the data with very complex queries. All these techniques rely on the MapReduce programming model which is currently one of the most used programming models used to process very large amount of data.

In the second part of this thesis we moved our focus to backward-chaining and proposed in Chapter 5 a technique to reduce the reasoning computation to

*`http://amartester.blogspot.nl/2007/04/bugs-per-lines-of-code.html`

be performed at query time. In Chapter 6, we showed how this technique can be implemented in a distributed setting and integrated in a simple SPARQL prototype.

Finally, in this last chapter we took a distance from our technical contribution trying to understand what makes our approaches "work" and what can limit their performance. We identify three crucial components behind the performance of our reasoning methods: the replication of the schema, the handling of data skew by ad-hoc data partitioning, and the engineering effort which is a required component to implement prototypes which are robust enough to work on a large scale.

Limitations. While we gave a concrete demonstration of web-scale reasoning, we have identified some important limitations in our approaches that require future work and that make in our opinion web-scale reasoning still an open issue.

First of all, in the context of forward-chaining our work relies on some assumptions (e.g. small size of the schema) that might not hold on future data. Also, our approach is unable to compute the closure if there is data which generates an explosion of the derivation. Usually such data are indications of syntactic or semantic mistakes, and a good practice would be to ignore it. However, our approach is not robust enough to identify it beforehand and future work is necessary to address this problem.

In the context of backward-chaining, our hybrid method performs well with selective queries (i.e. queries that retrieve and process relatively little amount of data). However, for larger queries the computation required by the backward-chaining algorithm becomes too expensive to be carried out in the interactive time. Also, in our implementation we have excluded the *sameAs* rules pointing out that they can be handled by maintaining a *sameAs* table as it is commonly done in other existing reasoners. Future work is necessary to identify a more efficient way to integrate them in the backward-chaining process since they are of great importance in our context.

The implementation of our hybrid-reasoning method in a distributed setting and integrating it into a SPARQL engine shows that the main limitation for efficient performance is in the transmission of intermediate data between the nodes and the synchronization mechanisms that are necessary to execute rules of the third and fourth type. While our caching strategy reduces the amount of data that is sent over the network, still the performance can be greatly improved if these issues are addressed with more efficient solutions. Another direction for future research consists of integrating inference in the

other operations that are normally operated with SPARQL like projections or filtering. Finally, an efficient mechanism to estimate the data cardinality is a fundamental piece to provide for a realistic integration of inference and SPARQL query processing.

Conclusions. In this thesis, we hope to have provided for a valid contribution to solve the very challenging problem of web-scale reasoning.

Looking back at the results presented in the previous chapters, we believe that we have shown that web-scale reasoning is indeed possible and that a distributed approach is a viable option to perform this task on an ever-growing amount of data, even some limitations still persist. Therefore, we argue that web-scale reasoning is no longer something that needs to be proved, but rather something that needs to be improved. To this end, future research is needed and requires multi-disciplinary expertise in order to solve all the issues that such a problem poses.

Part IV

Appendices

Appendix A

MapReduce Reasoning algorithms

We report on the MapReduce algorithms corresponding to the optimizations discussed in Sections 2.2 and 2.3 of Chapter 2. In Appendices A.1 and A.2 we report on the algorithms for RDFS and OWL reasoning respectively. For clarity purposes, we do not describe the algorithms in details but more on a higher level, using pseudo code instead of a real language. As usual, our pseudocode omits details that are not essential for human understanding of the algorithm, such as variable declarations, datatypes and some subroutines.*

A.1 RDFS MapReduce algorithms

We grouped the RDFS rules in four MapReduce jobs, to which we will refer to as `SUBPROP`, `DOMAINRANGE`, `SUBCLASS` and `SPECIAL_CASES`. These jobs are executed in sequence as described in Algorithm 17. The last job is not always executed because it refers to a special case that rarely occurs.

In the remaining of this section we discuss each of these reasoning algorithms.

*http://en.wikipedia.org/wiki/Pseudocode

SUBPROP applies rules 5 and 7, which concern sub-properties, and is reported in Algorithm 18. Since the schema triples are loaded in memory, these rules can be applied simultaneously. To avoid generation of duplicates, we follow the principle of setting as the tuple's key the triple's parts that are used in the derivation. This is possible because all inferences are drawn on an instance triple and a schema triple and we load all schema triples in memory. That means that for rule 5, we output as key the triple's subject while for rule 7 we output a key consisting of the subject and object. We add an initial flag to keep the groups separated since, later, we have to apply a different logic that depends on the rule. In case we apply rule 5, we output the triple's object as value, otherwise we output the predicate.

The reducer reads the flag of the group's key and applies to corresponding rule. In both cases, it first filters out the duplicates in the values. Then, it recursively matches the tuple's values against the schema and saves the output in a set. Once the reducer has finished with this operation, it outputs the new triples using the information in the key and in the derivation output set.

DOMAINRANGE applies rules 2 and 3, as shown in Algorithm 19. We use a similar technique as before to avoid generating duplicates. In this case, we emit as key the triple's subject and as value the predicate. We also add a flag so that the reducers know whether they have to match it against the domain or range schema. Pairs about domain and range will be grouped together if they share the same subject since the two rules might derive the same triple.

SUBCLASS applies rules 9, 11, 12, and 13, which are concerned with sub-class relations. The procedure, shown in Algorithm 20, is similar to the previous job with the following difference: during the map phase, we do not

Algorithm 17 RDFS overall algorithm

```
1   rdfs_reasoning(data) {
2     derived=apply_job(data, SUBPROP);
3     derived+=apply_job(data + derived, DOMAINRANGE);
4     derived=clean_duplicates(data, derived);
5     derived+=apply_job(data + derived, SUBCLASS);
6     if (derived_special_cases_no_loop(derived) == true)
7       derived+=apply_job(data + derived, SPECIAL_CASES);
8     if (special_cases_with_loop_occur(derived) == true)
9       derived+=rdfs_reasoning(data + derived);
10    return derived
11  }
```

Algorithm 18 RDFS sub-property reasoning (SUBPROP)

```
1   map(key, value):
2     if (subproperties.contains(value.predicate))
3       key = "1" + value.subject + "-" + value.object
4       emit(key, value.predicate)
5     if (subproperties.contains(value.object) && // for rule 5
6         value.predicate == "rdfs:subPropertyOf")
7       key = "2" + value.subject
8       emit(key, value.object)
9
10  reduce(key, values):
11    values = values.unique // filter duplicate values
12
13    switch (key[0])
14     case 1: // we are doing rule 7: subproperty inheritance
15       for (predicate in values)
16          // iterate over the predicates emitted in the map and collect superproperties
17         superproperties.add(subproperties.recursive_get(value))
18       for (superproperty in superproperties)
19         // iterate over superproperties and emit instance triples
20         emit(null, triple(key.subject, superproperty, key.object)
21     case 2: // we are doing rule 5: subproperty transitivity
22       for (predicate in values)
23         // iterate over the predicates emitted in the map, and collect superproperties
24         superproperties.add(subproperties.recursive_get(value))
25       for (superproperty in superproperties)
26         // emit transitive subproperties
27         emit(null, triple(key.subject, "rdfs:subPropertyOf", superproperty))
```

filter the triples which match with the schema but forward everything to the reducers instead. In doing so, this job also eliminates the duplicates against the input and we do not need to launch an additional job after this. The pseudocode of Algorithm 20 does not mention the computation of rules 12 and 13, because their execution is trivial.

SPECIAL_CASES refers to the special cases when rules 12 and 13 derive information which might fire rules 5, 7, 9 and 11. During the map phase, it loads in memory the subproperties of *rdfs:member* and the subclasses of *rdfs:Literal* and it performs the join between the triples in input with this schema. During the reduce, the job materializes the results of this join. Since this algorithm does not introduce any new challenges, we do not report the pseudocode.

Algorithm 19 RDFS domain and range reasoning (DOMAINRANGE)

```
1   map(key, value):
2    if (domains.contains(value.predicate)) then // for rule 2
3       key = value.subject
4       emit(key, value.predicate + "d")
5     if (ranges.contains(value.predicate)) then // for rule 3
6       key = value.object
7       emit(key, value.predicate +''r'')
8
9   reduce(key, values):
10    values = values.unique // filter duplicate values
11    for (predicate in values)
12      switch (predicate.flag)
13       case "r": // rule 3: find the range for this predicate
14         types.add(ranges.get(predicate))
15       case "d": // rule 2: find the domain for this predicate
16         types.add(domains.get(predicate))
17    for (type in types)
18       emit(null, triple(key, "rdf:type", type))
```

Algorithm 20 RDFS sub-class reasoning (SUBCLASS)

```
1   map(key, value):
2     if (value.predicate = "rdf:type")
3       key = "0" + value.subject
4       emit(key, value.object)
5     if (value.predicate = "rdfs:subClassOf")
6       key = "1" + value.subject
7       emit(key, value.object)
8
9   reduce(key, values):
10    values = values.unique // filter duplicate values
11
12    for (class in values)
13      superclasses.add(subclasses.get_recursively(class))
14
15    switch (key[0])
16     case 0: // we are doing rdf:type
17      for (class in superclasses)
18        if !values.contains(class)
19          emit(null, triple(key.subject, "rdf:type", class))
20     case 1: // we're doing subClassOf
21      for (class in superclasses)
22        if !values.contains(class)
23          emit(null, triple(key.subject, "rdfs:subClassOf", class))
```

Algorithm 21 Overall OWL reasoning algorithm

```
1   owl_readoning(data):
2     boolean first_time=true;
3     while (true) {
4       derived=rdfs_reasoning(data);
5       if (derived == null && first_time == false)
6          return data;
7       data= data + derived;
8
9       do { // Do fixpoint iteration for ter Horst rules
10        derived=apply_job(data, NOT_RECURSIVE_JOB);
11        derived+=apply_recursevily_job(data + derived, TRANSITIVITY_JOB);
12        derived=clean_duplicates(data, derived);
13        derived+=apply_recursevily_job(data + derived, SAME_AS_TRANSITIVITY_JOB);
14        derived=clean_duplicates(data, derived);
15        derived+=apply_job(data + derived, SAME_AS_INHERIT_1_JOB);
16        derived+=apply_job(data + derived, SAME_AS_INHERIT_2_JOB);
17        derived+=apply_job(data + derived, SAME_AS_INHERIT_3_JOB);
18        derived=clean_duplicates(data, derived);
19        derived+=apply_job(data + derived, EQUIVALENCE_JOB);
20        derived+=apply_job(data + derived, HAS_VALUES_JOB);
21        derived=clean_duplicates(data, derived);
22        derived+=apply_job(data + derived, SOME_ALL_VALUES_JOB);
23        derived=clean_duplicates(data, derived);
24        data= data + derived; }
25      while (derived != null);
26      first_time=false;
27    }
```

A.2 OWL MapReduce algorithms

OWL reasoning requires launching jobs in a loop until the rules stop deriving any conclusion. In Algorithm 21, we report the overall reasoning algorithm. It consists of a main loop where it first executes all the RDFS rules (see the previous section) and then executes the OWL rules until all rules do not derive anything anymore.

The OWL rules are implemented in ten MapReduce jobs. We describe each of them in the remainder of this section.

NOT_RECURSIVE_JOB executes rules 1, 2, 3, and 8 and is reported in Algorithm 22. These rules can be grouped and executed together. Their execution exploits the optimizations presented for the RDFS fragment. The instance triples are filtered during the map phase using the schema loaded in memory. The partitioning is done to avoid duplicates and the reduce function materializes the new triples.

TRANSITIVITY_JOB executes rule 4 and is reported in Algorithm 23. The map function filters out all the triples that do not have a transitive predicate by checking the input with the in-memory schema and it selects the triples which suit the possible join by checking their distance value. The reduce function simply loads the two sets in memory and returns new triples with distances corresponding to the sums of the combinations of distances in the input.

SAME_AS_TRANSITIVITY_JOB executes the logic of rule 7 to build the *sameAs* table and is reported in Algorithm 24. The sameAs triples are partitioned across nodes and the outgoing edges of a node are replaced by the incoming edge from the node with the lowest id, if such a node exists. This process is repeated until no edges can be replaced.

SAME_AS_INHERIT_*_JOBS is a sequence of three jobs that encode the logic of rule 11 and replace in the input triples every resource that appear in the *sameAs* table with its corresponding canonical representation. The first job counts the resources and identifies the most popular one. This is done to prevent load balancing issues in the following jobs. The second and third jobs perform the replacements. The replacements are done by deconstructing the statements and executing the join between the *sameAs* table and the input triples in the classical way. We do not report the pseudocode of these jobs because the technique is completely analogous to the data decompression technique described in Chapter 3 with the only difference being that here we perform the join against the *sameAs* table and not the dictionary.

EQUIVALENCE_JOB executes rules 12 and 13. The algorithm is not reported because its implementation is straightforward. These rules work with schema triples and therefore do not present any particular challenges. The corresponding job loads the schema in memory and derives the information without any duplicate derivations using the RDFS optimizations.

HAS_VALUE_JOB executes rules 14. Similarly, this algorithm performs the join between the schema triples in the nodes' main memory and filters the instance triples during the map phase. The reduce phase simply materializes the derivation while eliminating the eventual duplicates between the derivation.

Algorithm 22 OWL not recursive rules (NOT_RECURSIVE_JOB)

```
1   map(key, triple):
2    if (functional_properties.contains(triple.predicate)) then //Rule 1
3       emit({'F',triple.subject,triple.predicate}, triple.object);
4    if (inverse_functional_properties.contains(triple.predicate)) then //Rule 2
5       emit({'F',triple.object,triple.predicate}, triple.subject);
6    if (symmetric_properties.contains(triple.predicate)) then //Rule 3
7       emit({'S',triple.subject,triple.object}, triple.predicate);
8    if (inverse_properties.contains(triple.predicate)) then //Rule 8
9       emit({'I',triple.subject,triple.object}, triple.predicate);
10
11  reduce(key, values):
12     switch (key[0])
13        case 'F' : for (value in values) do //Process rule 1 and 2
14                      synonyms.add(value)
15                   for (synonym in synonyms) do
16                      emit('synonyms.min_value, owl:sameAs, synonym);
17       case 'S' : for (value in values) do //Process rule 3
18                      emit(key.object, value, key.subject);
19       case 'I' : for (value in values) do //Process rule 8
20                      inverse_property = inverse_properties.get(value)
21                      emit(key.object, inverse_property, key.subject);
```

SOME_ALL_VALUES_JOB executes rules 15 and 16 and is reported in Algorithm 25. It aims at reducing the size of the partitions by performing the joins with the schema triples as soon as possible. We first perform the join between the two schema triples. Then, we perform the join between the above and either `(u p x)` or `(x rdf:type w)`, calculating `(v owl:someValuesFrom w)` ⋈ `(v owl:onProperty p)` ⋈ `(u p x)` and `(x rdf:type w)` ⋈ `(v owl:onProperty p)` ⋈ `(v owl:someValuesFrom w)`. Now, we have all possible bindings for x and p. Thus, we can partition on `(xv)` and perform the join during the reduce phase. The algorithm for `(owl:allValuesFrom)` is analogous to this one and not reported for conciseness.

Algorithm 23 OWL transitivity closure, p rule 4 (TRANSITIVITY_JOB)

```
1  map(key, triple):
2   n = job_config.get_current_step();
3   if (key.step = 2^(n - 2) || key.step = 2^(n -1)) then
4    emit({triple.predicate, triple.object}, {flag=L, key.step, triple.subject});
5   if (key.step > 2^(n-2) then
6    emit({triple.predicate, triple.subject}, {flag=R, key.step, triple.object});
7
8  reduce(key, iterator values):
9    for(value in values) do
10     if (value.flag = 'L')
11       leftSide.add({key.step, value.subject})
12     else
13       rightSide.add({key.step, value.object})
14   for(leftElement in leftSide)
15     for(rightElement in rightSide)
16       newKey.step = leftElement.step + rightElement.step //distance new triple
17       emit(newKey,triple(leftElement.subject, key.predicate, rightElement.object));
```

Algorithm 24 (SAME_AS_TRANSITIVITY_JOB): OWL build sameAs table, p rule 7

```
1  map(key,edge):
2    emit(edge.from,edge.to);
3    emit(edge.to,edge.from);
4
5  reduce(key,values):
6    toNodes.empty(); // edges to other nodes
7    foundReplacement = false
8    for(value in values)
9      if (value < key)
10       if (foundReplacement)
11         toNodes.add(key);
12       foundReplacement = true;
13       key = value;
14     else if (value > key) toNodes.add(value);
15   for(to in toNodes)
16     emit(null,{key,to});
```

Algorithm 25 OWL someValuesFrom and allValuesFrom reasoning, p rule 15 and 16 (SOME_ALL_VALUES_JOB)

```
1  map(key, triple):
2     joinSchema = join on the subject between someValuesFrom and onProperties triples
3     if (triple.predicate == "rdf:type")
4       if (triple.object in joinSchema.someValuesFromObjects)
5            entries = joinSchema.getJoinEntries(triple.object)
6            for (entry in entries)
7              emit({entry.p,triple.subject}, {type=typetriple, resource=entry.
                    onPropertySubject});
8     else if (triple.predicate in joinSchema.onPropertiesSet)
9       emit({triple.predicate,triple.object}, {type=generictriple, resource=triple.subject
             });
10
11 reduce(key, values):
12    types.clear(); generic.clear();
13     for (value in values)
14       if (value.type = typetriple) types.add(value.resource)
15       else generic.add(value.resource)
16     for (v in types)
17       for (u in generic)
18         emit(null, triple(u, "rdf:type", v));
```

Appendix B

SPARQL queries

B.1 Queries for Yahoo! use-case

Query 1

```
Select (count(?subj) as ?freq) (min(?subj) as ?example)
 ?CSct ?CSdi ?CSmx ?CSmi { {
  Select ?subj (count(?subj) as ?CSct)
         (count(distinct ?prop) as ?CSdi)
 (max(?prop) as ?CSmx) (min(?prop) as ?CSmi) {
     ?subj ?prop ?obj }
     Group By ?subj } }
Group by ?CSct ?CSdi ?CSmx ?CSmi
Order By desc(?freq)
Limit 10000
```

Query 2

```
SELECT  ?p (COUNT(?s) AS ?count )
{ ?s ?p ?o } GROUP BY ?p ORDER BY ?count
```

Query 3

```
SELECT  ?C (COUNT(?s) AS ?count )
{ ?s a ?C } GROUP BY ?C ORDER BY ?count
```

B.2 BSBM queries

Every query in the BSBM-BI benchmark contains some parameters that are replace by values from the dataset. In the evaluation section of chapter 4, we have replaced the parameters with the values reported below.

Query	Parameter	Value
1	Country1	`http://downlode.org/.../countries#GB`
	Country2	`http://downlode.org/.../countries#US`
2	Product	`http://.../dataFromProducer1/Product5`
3	ConsecutiveMonth_0	2008-03-11T00:00:00
	ConsecutiveMonth_1	2008-04-11T00:00:00
	ConsecutiveMonth_2	2008-05-11T00:00:00
4	ProductType	`http://.../instances/ProductType2`
5	ProductType	`http://.../instances/ProductType2`
6	Producer	`http://.../dataFromProducer1/Producer1`
7	Country	`http://downlode.org/.../countries#GB`
	ProductType	`http://.../instances/ProductType2`
8	ProductType	`http://.../instances/ProductType2`

B.3 LUBM queries

N.	Query
1.	(?x :takesCourse :University0) (?x rdf:type :Student)
2.	(?x :publicationAuthor :AssistantProfessor0) (?x rdf:type :Publication)
3.	(?x :worksFor :University0) (?x rdf:type :Professor) (?x :name ?y1) (?x :emailAddress ?y2) (?x :telephone ?y3)
4.	(?x :memberOf :University0) (?x rdf:type Person)
5.	(AssociateProfessor0 :teacherOf ?y) (?y rdf:type :Course) (?x :takesCourse ?y) (?x rdf:type :Student)
6.	(?y :subOrganizationOf :University0) (?y rdf:type :Department) (?x :memberOf ?y) (?x rdf:type :Student) (?x :emailAddress ?z)
7.	(?x :takesCourse :GraduateCourse0) (?x rdf:type :Student)
8.	(?x :subOrganizationOf :University0) (?x rdf:type :ResearchGroup)
9.	(?y :subOrganizationOf :University0) (?y rdf:type :Department) (?x :worksFor ?y) (?x rdf:type :Chair)
10.	(:University0 :hasAlumnus ?x) (?x rdf:type :Person)

Bibliography

[1] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 411–422. VLDB Endowment, 2007.

[2] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: Run-time Optimization of XQueries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 615–626. ACM, 2009.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[5] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of the International World-Wide Web Conference*, pages 41–50, 2010.

[6] H. E. Bal, J. Maassen, R. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, T. Kielmann, F. J. Seinstra, and C. J. H. Jacobs. Real-World Distributed Computing with Ibis. *IEEE Computer*, 43(8):54–62, 2010.

[7] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.

[8] D. Battré, A. Höing, F. Heine, and O. Kao. On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores. In *Proceedings of the VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2006.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[10] BigData. `http://www.bigdata.com/bigdata/blog/`, 2012.

[11] Billion Triple Challenge Website. `http://challenge.semanticweb.org`, 2012.

[12] Bio2RDF. `http://bio2rdf.org`, 2012.

[13] B. Bishop and S. Bojanov. Implementing OWL 2 RL and OWL 2 QL rule-sets for OWLIM. In *Pro. of the 8th International Workshop OWL: Experiences and Directions*, 2011.

[14] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[15] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In *Spinning the Semantic Web*, pages 197–222, 2003.

[16] M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the International World-Wide Web Conference*, 2004.

[17] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990. ISBN 3-540-51728-6.

[18] DAS-3. `http://www.cs.vu.nl/das3`, 2012.

[19] data.gov.uk. `http://data.gov.uk/linked-data`, 2012.

[20] DBPedia. `http://dbpedia.org`, 2012.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–147, 2004.

[22] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of Reasoners for large Ontologies in the OWL 2 EL Profile. *Semantic Web Journal*, 2(2): 71–87, 2011.

[23] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2009. ISBN 978-3-642-04328-4.

[24] FactForge. `http://www.factforge.com`, 2012.

[25] Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable Distributed Ontology Reasoning Using DHT-based Partitioning. In *Proceedings of the Asian Semantic Web Conference (ASWC)*, 2008.

[26] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto. RDF Compression: Basic Approaches. In *Proceedings of the International World-Wide Web Conference*, pages 1091–1092. ACM, 2010. ISBN 978-1-60558-799-8.

[27] M. Gallego, J. Fernández, M. Martínez-Prieto, and P. Fuente. An Empirical Study of Real-World SPARQL Queries. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at WWW 2011*, 2011.

[28] S. Ganguly, P. Gibbons, Y. Matias, and A. Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–281. ACM, 1996.

[29] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, volume 2, pages 1414–1425. VLDB Endowment, 2009.

[30] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3:158–182, 2005.

[31] Hadoop. `http://hadoop.apache.org`, 2012.

[32] O. Hartig, C. Bizer, and J. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 293–309. Springer, 2009.

[33] P. Hayes, editor. *RDF Semantics*. W3C Recommendation, Feb. 2004.

[34] S. Heinz, J. Zobel, and H. E. Williams. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, 20:192–223, 2002.

[35] A. Hogan, A. Harth, and A. Polleres. Scalable Authoritative OWL Reasoning for the Web. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

[36] A. Hogan, J. Pan, A. Polleres, and S. Decker. SAOR: Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples. In *Proceedings of the International Semantic Web Conference (ISWC)*. Springer Berlin / Heidelberg, 2010.

[37] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[38] M. F. Husain, P. Doshi, L. Khan, and B. Thuraisingham. Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. In M. G. Jaatun, G. Zhao, and C. Rong, editors, *Cloud Computing*, volume 5931, chapter 72, pages 680–686. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-10664-4.

[39] Z. Ives and N. Taylor. Sideways Information Passing for Push-Style Query Processing. In *Proceedings of the International Conference on Data Engineering*, pages 774–783. IEEE, 2008.

[40] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.

[41] R. Kader, M. Van Keulen, P. Boncz, and S. Manegold. Run-time Optimization for Pipelined Systems. In *Proceedings of the IV Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.

[42] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2008.

[43] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM – a Pragmatic Semantic Repository for OWL. In *Proceedings of the Conference on Web Information Systems Engineering (WISE) Workshops*, pages 182–192, 2005.

[44] V. Kolovski, Z. Wu, and G. Eadon. Optimizing Enterprise-scale OWL 2 RL Reasoning in a Relational Database System. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 436–452. Springer, 2010.

[45] S. Kotoulas, E. Oren, F. van Harmelen, and F. van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *Proceedings of the International World-Wide Web Conference*, pages 531–540, 2010.

[46] G. Ladwig and T. Tran. Linked Data Query Processing Strategies. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 453–469. Springer, 2010.

[47] Large triple stores wiki page. `http://esw.w3.org/topic/LargeTripleStores`, 2012.

[48] LarKC deliverable 5.2.2. `http://hadoop.apache.org`, 2012.

[49] LDSR. `http://www.ontotext.com/ldsr/`, 2012.

[50] K. Lee, J. H. Son, G.-W. Kim, and M.-H. Kim. Web document compaction by compressing uri references in RDF and OWL data. In *ICUIMC*, pages 163–168, 2008.

[51] Linked Life Data. `http://www.linkedlifedata.com`, 2012.

[52] D. McGuinness, F. Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C recommendation*, 10:2004–03, 2004.

[53] B. S. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang. URL Forwarding and Compression in Adaptive Web Caching. In *In Proc. of IEEE Infocom*, pages 670–678, 2000.

[54] R. Mutharaju, F. Maier, and P. Hitzler. A MapReduce Algorithm for EL+. In *Proceedings of the 23rd International Workshop on Description Logics (DL2010), Waterloo, Canada*, 2010.

[55] H. Nagumo, M. Lu, and K. Watson. Parallel Algorithms for the Static Dictionary Compression. In *Proc. IEEE Data Compression Conf*, pages 162–171, 1995.

[56] W. Nejdl. Recursive Strategies for Answering Recursive Queries-The RQA/FQI strategy. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 43–50, 1987.

[57] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proceedings of the International Conference on Data Engineering*, pages 984–994. IEEE, 2011.

[58] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 627–640. ACM, 2009.

[59] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, 2010.

[60] A. Newman, Y. Li, and J. Hunter. Scalable Semantics the Silver Lining of Cloud Computing. In *Proceedings of the 4th IEEE International Conference on eScience*, 2008.

[61] M. K. Nguyen, C. Basca, and A. Bernstein. B+Hash Tree: Optimizing query execution times for on-Disk Semantic Web data structures. In A. Fokoue, T. Liebig, and Y. Guo, editors, *Proceedings Of The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 96–111, November 2010.

[62] Official statistics of Linked Data Website. `http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics`, 2012.

[63] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM, 2008.

[64] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen. Marvin: Distributed reasoning over large-scale Semantic Web data. *Journal of Web Semantics*, 7(4):305–316, 2009.

[65] H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient Query Answering for OWL 2. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 489–504. Springer, 2009.

[66] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[67] P. Ravindra, V. Deshpande, and K. Anyanwu. Towards scalable RDF graph analytics on MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, page 5. ACM, 2010.

[68] M. Salvadores, G. Correndo, S. Harris, N. Gibbins, and N. Shadbolt. The Design and implementation of minimal RDFS backward reasoning in 4store. *The Semanic Web: Research and Applications*, pages 139–153, 2011.

[69] A. Schätzle and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *SWIM: the 3th International Workshop on Semantic Web Information Management*, 2011.

[70] A. Schlicht and H. Stuckenschmidt. Peer-to-peer reasoning for interlinked ontologies. *International Journal of Semantic Computing*, 4(1):27–58, 2010.

[71] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 82–97, 2008.

[72] C. Seitz and R. Schönfelder. Rule-Based OWL Reasoning for Specific Embedded Devices. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 237–252, 2011.

[73] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. *PVLDB*, 1 (2):1553–1563, 2008.

[74] R. Soma and V. Prasanna. Parallel Inferencing for OWL Knowledge Bases. In *International Conference on Parallel Processing*, pages 75–82, 2008.

[75] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[76] P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2010.

[77] Swoogle. `http://swoogle.umbc.edu`, 2012.

[78] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2–3):79–115, 2005.

[79] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive-A Warehousing Solution Over a Map-Reduce Framework. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2009.

[80] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse using Hadoop. In *Proceedings of the International Conference on Data Engineering*, pages 996–1005, 2010.

[81] Uniprot. `http://www.uniprot.org`, 2012.

[82] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning using MapReduce. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2009.

[83] J. Urbani, S. Kotoulas, J. Maassen, N. Drost, F. Seinstra, F. van Harmelen, and H. Bal. WebPIE: a Web-scale Parallel Inference Engine. 1st prize at the 3rd IEEE SCALE challenge at CCGrid, 2010.

[84] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL reasoning with MapReduce: calculating the closure of 100 billion triples. In *Proceedings of the European Semantic Web Conference (ESWC)*, 2010.

[85] J. Urbani, J. Maaseen, and H. Bal. Massive Semantic Web data compression with MapReduce. In *Proceedings of the 1st MapReduce workshop at HPDC '10*, 2010.

[86] J. Urbani, S. Kotoulas, J. Maassen, F. V. Harmelen, and H. Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Journal of Web Semantics*, 10(0):59 – 75, 2012. ISSN 1570-8268.

[87] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal. Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 2012. ISSN 1532-0634.

[88] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, 23(4):733–742, 1976.

[89] L. Vieille. A Database-Complete Proof Procedure Based on SLD-Resolution. In *ICLP*, pages 74–103, 1987.

[90] W3C Recommendation. OWL Web Ontology Language Overview: Why OWL?, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.2`.

[91] W3C Recommendation. OWL 2 Web Ontology Language: Document Overview, 27 October 2009. Available at `http://www.w3.org/TR/owl2-overview/`.

[92] W3C Recommendation. OWL 2 Web Ontology Language Profiles, 27 October 2009. Available at `http://www.w3.org/TR/2009/REC-owl2-profiles-20091027`.

[93] W3C Recommendation: RDF Primer. `http://www.w3.org/TR/rdf-primer/`, 2012.

[94] J. Weaver and J. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *Proceedings of the International Semantic Web Conference (ISWC)*, October 2009.

[95] WebPIE. `http://www.cs.vu.nl/webpie`, 2012.

[96] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, volume 1, pages 1008–1019. VLDB Endowment, 2008.

[97] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 1040. ACM, 2007.

[98] Y. Ye and P. Cosman. Dictionary design for text image compression with JBIG 2. *IEEE Transactions on Image Processing*, 10(6):818–828, 2001.

[99] J. Zobel, S. Heinz, and H. E. Williams. In-memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80:2001, 2001.

Samenvatting

Over Web-scale Reasoning

Het semantische web is een uitbreiding van het huidige wereldwijde web. In het semantische web kan de betekenis van informatie door machines geïnterpreteerd worden en de data worden opgeslagen als een verzameling van subject-predicaat-object verbanden.

Op dit moment zijn er miljarden van dergelijke verbanden publiek beschikbaar op het web. Deze verbanden beschrijven een zeer breed scala van informatie: van biomedische informatie tot informatie afkomstig van regeringen. Hierbij worden URIs vaak gebruikt om concepten ondubbelzinnig te kunnen identificeren en hergebruik in gedistribueerde omgevingen, zoals het wereldwijde web, te stimuleren.

Een van de voordelen van het opslaan van informatie met gebruik van semantische web technologieën is dat computers over de data kunnen redeneren en nieuwe informatie kunnen afleiden. Dit proces, ook wel reasoning genoemd, wordt in toenemende mate uitdagender, vanwege de exponentiële groei van beschikbare data op het web. In het begin van 2009 werd het aantal van dergelijke verbanden op het semantische web geschat op 4,4 miljard. Een jaar later is de grootte van het web verdrievoudigd naar 12 miljard verbanden en de huidige trend wijst er op dat een dergelijke groei nog steeds plaatsvindt.

De onderzoeksvraag die in dit proefschrift beantwoord wordt, is: Hoe kunnen we door de inzet van reasoning op een parallel en gedistribueerd systeem de resultaten van zoekopdrachten over zeer grote hoeveelheden data verrijken?

Het proefschrift bestaat uit twee delen: in het eerste deel vindt reasoning plaats door het toepassen van een verzameling van regels over de gehele invoerdata, teneinde elke mogelijke conclusie over de gehele invoerdata te verkrijgen. Het tweede deel behandelt een andere benadering waarin enkel die conclusies verkregen worden die relevant zijn voor de zoekopdrachten van de gebruikers.

In het eerste deel van het proefschrift is het MapReduce programmeermodel gebruikt om reasoning op een grootschalige manier uit te voeren en zodoende goede prestaties te behalen. In het tweede hoofdstuk wordt een reeks van op MapReduce gebaseerde reasoning-algoritmes geïntroduceerd. In het derde hoofdstuk gebruiken we hetzelfde programmeermodel om de invoerdata te comprimeren in een compactere vorm, waardoor er efficiënter gerekend kan worden met de invoerdata. In het vierde hoofdstuk gebruiken we Pig, een taal die voortbouwt op MapReduce, om grote SPARQL zoekopdrachten te coderen. Op deze manier voorzien we in een compleet systeem voor het afleiden van nieuwe informatie en het doen van zoekopdrachten, waarbij eenzelfde systeemarchitectuur en programmeermodel wordt gebruikt.

In het tweede deel verplaatsen we de aandacht naar een vorm van reasoning die wordt uitgevoerd wanneer een gebruiker een zoekopdracht uitvoerd in een kennisbank. In een dergelijke situatie kan MapReduce niet worden gebruikt aangezien de uitvoering van een dergelijke MapReduce taak een lange wachttijd oplevert. Daarom

introduceren we een nieuwe hybride reasoning-techniek waarmee we alleen een klein deel van de nieuwe informatie vooraf afleiden. Deze nieuwe informatie wordt tijdens het uitvoeren van de zoekopdracht gebruikt om de rekentijd van de zoekopdracht in te korten.

In het vijfde hoofdstuk analyseren we onze techniek van hybride reasoning vanuit een theoretisch perspectief en verifiëren we of de aanpak correct en volledig is. In het daaropvolgende hoofdstuk, hoofdstuk 6, beschrijven we een gedistribueerd en parallel prototype van deze techniek en analyseren we de prestaties op het DAS-4 cluster met een standard benchmark.

In het laatste hoofdstuk van dit proefschrift leiden we een aantal principes, die we als "wetten" beschouwen, af uit de technische bijdragen die in de vorige hoofdstukken zijn gepresenteerd. Deze wetten zijn aantoonbaar geldig voor de huidige data en zijn ook verantwoordelijk voor de resultaten die we behaald hebben in onze experimenten. De wetten kunnen worden gebruikt om een beter begrip te krijgen van de eigenschappen van het huidige onderwerp van reasoning op de schaal van het web en kunnen gebruikt worden om verder onderzoek over dit onderwerp te bevorderen.