MASTER THESIS

RDFS/OWL reasoning using the MapReduce framework

Jacopo Urbani

July 2, 2009

Supervisor:
Eyal Oren

Second reader:
Frank van Harmelen

Vrije Universiteit - Faculty of Sciences
Dept. of Computer Science
De Boelelaan 1081, 1081 HV - Amsterdam

Acknowledgments

The value of the master thesis goes beyond its scientific content since it summarizes the work of five years of education. In my particular case this thesis symbolically represents all the years spent between Venice, Vienna and Amsterdam, with all the gained amount of experience and personal development.

My master project took more or less six months. This document reports only the final outcome, leaving out everything that is behind it. My thesis is also made by many emails, meetings, hours spent in reading articles, testing and debugging. During this time I had the honor to work with many great persons and I want to thank all of them, for their support and patience.

More in particular my first thanks go to Eyal, my supervisor, that helped me through all the development of this thesis, whatever the problem was. He tried to teach me two of the nicest virtues in the scientific world: conciseness and clarity. I hope I have learned them well. This thesis received also a crucial help from Spyros Kotoulas. Without his observations and critics this work could not be done.

I also would like to thank Frank van Harmelen who offered me a job and financial support for my work. Without his support, my life would have been much more complicated.

A special thanks goes to all the users of the DAS3 cluster, who patiently accepted my constant abuses of the cluster when I was running the tests and, more in general, to all the researchers who wrote all the literature or programs that are related to this thesis. It is because of their previous work that I could do mine.

My thanks go also to some people outside the university starting from my family for all the precious support and advices they gave me. I also would like to thank my friends, that kept me away from my work once a while.

At last, but not least, my immense gratitude goes to my girlfriend Elsbeth, who let me work also in the weekends and gave me the moral support I needed. This thesis is dedicated to her.

Contents

Chapter 1

Introduction

In 1994 Tim Berners-Lee introduced the concept of the Semantic Web. The Semantic Web is a extension of the traditional web that can be roughly described as an attempt of injecting a semantic meaning to all of the information that populate the web [28]. Nowadays the Semantic Web can count on a solid base of literature and developed software.

In the Semantic Web world the information is encoded in some specific languages where the associated semantics can be understood not only by human beings as it is with the commonly spoken languages but also from the computers. People communicate to each others using complex languages like, for example, the English language. Machines are unable to understand such language. For example the sentence "Alice eats an apple" contained in a web page is for a machine nothing more than a mere sequence of bytes. Instead a person can read this sentence and derive some new information like that "Alice" is a person, that "eats" is a word that indicate the action of eating and that "apple" is a fruit. All this information cannot be derived by a machine because it cannot catch and manipulate the semantics contained in the sentence.

Semantic Web addresses this problem by introducing a set of standards and tools organized in a stack called Semantic Web stack. The aim of Semantic Web is to provide a set of technologies in such a way that machines can somehow process the information using the semantics contained in it. In the Semantic Web machines can retrieve certain information more efficiently than before or being able to derive new information from an existing data set[28].

The process of deriving new information is called reasoning and there are already some programs that do this in a efficient way [29] [3] [14]. However in the last years the amount of information in the Semantic Web has considerably increased making the reasoning process a data intensive problem where the physical constraints of a single machine are a notable limitation. Scalability is an essential feature because the Semantic Web is based on the top of the traditional web and experience showed how rapidly the amount of available information has increased on Internet. If Semantic Web wants to continue its ascensions in usage and popularity it must provides tools that can handle a large

amount of information.

> In this thesis I will address the problem of reasoning over a large amount of data using a distributed system, and, more in details, using the MapReduce model[4] offered by the Hadoop framework. The hypothesis is that the reasoning process can be efficiently translated into one or more MapReduce jobs and the purpose of this work is to verify it.

In order to prove this hypothesis we designed and evaluated some algorithms that do RDFS [8] and OWL [27] reasoning with the MapReduce programming model. In this document we describe these algorithms in detail along with the performances we have obtained in our tests.

1.1 Outline

The rest of this chapter focuses on the advantages and disadvantages of using a distributed system. The other chapters are organized as follows. Chapter 2 describes some ground terms and the technologies used in this work so that the reader has enough knowledge to understand the rest. In case the reader is already familiar with MapReduce and the Semantic Web technologies this chapter can be easily skipped. Chapter 3 contains an overview of some related work and of some already existing reasoners. Chapter 4 presents an algorithm that compresses the data using dictionary encoding. Data compressiong is a technical problem that, though is not strictly related to reasoning, is necessary for our final purpose. Chapter 5 discusses about the design and implementation of an algorithm that does RDFS reasoning. Chapter 6 does the same about OWL reasoning. Chapter 7 reports the results obtained with our implementation and an analysis of the performances. At last, chapter 8 reports the conclusions and some possible future extensions of this work.

1.2 Parallel and distributed reasoning

Today just thinking of storing all the web information on one machine is pure science fiction. The resources of one machine are way too small to handle even a small fraction of the information in the web. Semantic Web is probably in the same situation than the former web at its beginning. It is necessary to move from a single environment perspective to a distributed setting in order to exploit the Semantic Web on a global scale.

With a distributed system we overcome the limitation of physical hardware constraints, but other problems are introduced making this problem not trivial to solve. In general we are able to exploit the advantages of a distributed system only if we can partition the input so that the single nodes can work without communicating to each others. If there is a strong correlation between the data we cannot split the input and the nodes cannot operate independently. The

communication between the nodes generates overhead with the consequence of worsening the performances.

Unfortunately the data in Semantic Web is strongly correlated and the reasoning process worsens it because the derived information connects the data even more than before. This consideration does not play in favor of a distributed system but we still aim to find way to partition efficiently the data so that we can exploit the advantage of parallelism.

Another problem that arise if we use a distributed system is load balancing. We must take care that the workload is equally distributed between the nodes, otherwise some of them will work much more than the others and we will miss all the advantages in having a parallelization of the computation.

The MapReduce programming model [4] is described in detail in section 2.6. Here we will simply sketch it as a programming model where the computation is defined in *jobs* and every job consists in two phases: *map* and *reduce*. *Map* is a function that creates some partitions over the input data. *Reduce* is a function that processes each of these partitions one by one.

The methodology of first partitioning and then further process the partitions provides an high level of parallelism. The main advantage of using a MapReduce framework is that we can concentrate on the logic of the program (design the map and reduce functions) without worrying so much about the execution and everything that concerns technical details.

Encoding reasoning as a MapReduce job means solving two problems. The first problem consists in how to partition the data in order to reason over it and this is not trivial because of the high correlation. The second problem consists in how to process the partitions and eventually infer new triples.

The first problem is solved by writing an appropriate *map* algorithm. Analogously the second problem implies writing a proper *reduce* algorithm. After we have defined them, the framework will execute the job in a pseudo-transparent way with an high degree of parallelization.

Chapter 2

Background

In this chapter we will describe the technologies that are used in this work with the purpose of providing a basic and common background to ease the understanding of the rest of the document.

Basically the problem we deal with consists in processing some data in input deriving some new information out of it. We call this process "reasoning" over the data. The data in input can be encoded either in RDF or OWL and the reasoning depends on which language the data is encoded in.

In section 2.1 we outline more formally what the term reasoning means for us giving an overview of the types of reasoning that is possible to do. In section 2.2, we provide a basic description of the Semantic Web illustrating the Semantic Web stack and the XML language. Section 2.3 contains a brief description of RDF/RFDS while section 2.5 does the same for the OWL language. In section 2.6 we describe the MapReduce programming model with a simple example. At last, in section 2.7 we report a brief description of the Hadoop framework that is the framework used for the implementation and the evaluation of our approach.

2.1 Reasoning

Reasoning can be roughly defined as a process from which we derive new information using an already existing set of data. In general reasoning can be divided in either deductive or inductive reasoning.

In this thesis we will discuss only of deductive reasoning. In deductive reasoning if certain premises are true then also a certain conclusion must be true [12]. Let's make a simple example, using a RDF construct.

Suppose we have two triples:

```
Alice isa Student .
Student subclassof Person .
```

An example of deductive reasoning could be: *if someone is a something (first premise) and this something is a subclass of something else (second premise)*

then someone is a something else.

Following this deductive process we are able to derive the new information

```
Alice isa Person .
```

We can represent the reasoning through rules. A rule is made by a set of premises and one or more conclusions. An example of rule is

```
if   A type B
and  B subclassof C
then A type C
```

In the example above, the deductive process that derived that Alice is a Person can be condensed in this last rule. In general whenever we find some information that matches the rule's premises we can derive the information contained in the conclusion.

Reasoning is also divided in two other categories, backward and forward reasoning, depending on which starting point we take as input. If we take the existing information as starting point and we want to derive all the possible statements then we are doing forward reasoning. In the example above we applied forward reasoning because we started from two existing triples and we checked if the premises were true so that we could derive the new information.

Backward reasoning works as follows. We pick one conclusion (like Alice isa Person) and see if the premises for this conclusion hold. This process is done recursively till we "walk back" to the input data. If the input data confirms the premises then we can successfully assert that the information is true.

There are advantages and disadvantages in choosing either forward or backward reasoning. The first is normally used when we need to materialize every statement out of an existing input. The second is mainly used for queries or verify if some conclusions are correct.

In this thesis when we talk about reasoning we mean forward deductive reasoning encoded as a set of rules. We can see the reasoner implemented in this thesis as a program that applies a set of rules continuously to a set of data till nothing else can be derived anymore.

We distinguish two different reasoners, one that exploits the constructs of RDF Schema and one that works with the data in OWL format. The first is referred as RDFS reasoner while the second as OWL reasoner. The first reasoner is simpler than the second in terms of complexity.

2.2 Semantic Web and XML

Semantic Web is a set of tools and languages that are composed in the so called Semantic Web stack [28]. This stack is reported in Figure 2.1.

As we can see from the figure the Semantic Web uses XML as a standard language [5]. What HTML is for the web XML is for the Semantic Web. Practically all the information in Semantic Web is encoded in XML. The choice of using XML to represent the information is due to two reasons:

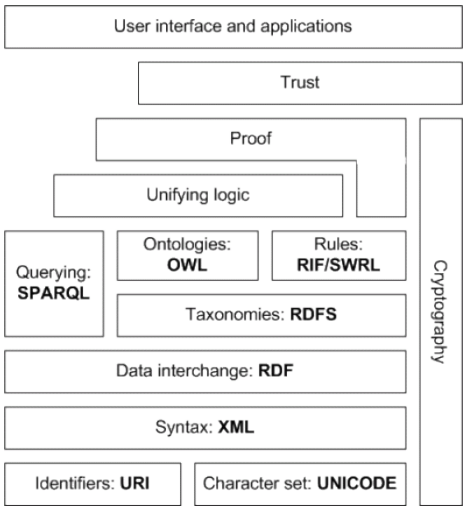

Figure 2.1: The semantic web stack

- with XML we can encode a wide range of data and this is a necessary condition since in the Semantic Web data can be of any possible form;
- XML is already widely used. There are already many parsers and writers that make the transmission of the information possible between different systems.

We report below a small fragment of a possible XML document:

```
<order>
  <productId>p1</productId>
  <amount>12</amount>
  <company>X</company>
</order>
```

Using XML we are able to structure the information in a tree using some specific tags. In this example we can impose for example that an order (defined with the tag <*order*>) must contain a product and an amount and not viceversa. The structure can be defined either with the DTD (Document Type Definition) or with the W3C XML Schema.

XML does not provide a semantics for the tag nesting [22]. For example in the reported example the meaning between <*order*> and <*company*> is ambiguous. The information contained in that small fragment could be intended as:

> The company *X* made an order of *12* units of product *p1*.

or

> The order should be requested to company *X* and it consists of *12* units of product *p1*.

The comprehension of the document is left to the context where it is used. The different parties who exchange XML data must beforehand agree on the semantics of the data they exchange [5]. This operation can be easily done if there are two (or a small number) of fixed parties but on the web where there are many actors this operation becomes difficult.

2.3 RDF

RDF consists in a data model released by W3C on 1999[1]. With RDF the information is encoded in statements, where each statement consists in a triple of the form *object attribute value* [15].

Every triple can be seen as a small sentence composed by a subject, a predicate and an object. An example of statement can be the triple "Alice eats apple" where Alice is the subject, eats is the predicate and apple is the object. The same triple can also be seen in terms of the relation object-attribute-value.

In RDF the elements that compose a triple are called resources[15]. Resources can be either URIs, literal or blank nodes. In a typical Semantic Web setting Alice could be the URIs that point to the home page of that person. The choice of using URIs as standard identifiers instead of simple text is due mainly because the URI nomenclature is standard in the traditional web and because an URI is supposed to be unique over the web and therefore ideal to identify unique entities. In XML and RDF URIs are often reported in the abbreviated form *namespace:fragment*. For example the URI *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* is often abbreviated in *rdf:type*.

An RDF statement can be serialized in different formats. The most common formats are RDF/XML[2], N3[3], N-triples[4] and Turtle[5]. The first is an XML format. In this way RDF inherits all the advantages about the syntax interoperability of XML. However RDF is independent by XML and can be represented in other formats that share nothing with XML (like N3 or N-Triples) [22]. Below we report a small example of the same RDF information encoded in different formats.

RDF/XML:

```
<rdf:Description rdf:about="http://www.student.vu.nl/~jui200">
  <hasName>Jacopo Urbani</hasName>
</rdf:Description>
```

[1] http://www.w3.org/TR/PR-rdf-syntax/
[2] http://www.w3.org/TR/rdf-syntax-grammar/
[3] http://www.w3.org/DesignIssues/Notation3
[4] http://www.w3.org/TR/rdf- testcases/#ntriples
[5] http://www.w3.org/TeamSubmission/turtle/

```
N-Triples:

<http://www.student.vu.nl/~jui200>
<hasName>
"Jacopo Urbani" .
```

An RDF document is made by a set of statements. Using RDF Schema we are able to define a vocabulary over the data model providing a form of semantics that is accessible at a machine level.

2.4 RDF Schema

As we have seen before, RDF is a standard language that allows us to encode the information using statements made of triples. The RDF Schema (abbreviated RDFS) is an extension of RDF that allows the users to define the vocabulary used in RDF documents [2]. Through RDF Schema we are able to define some special relations between the resources which have a unique meaning. One example is "rdfs:subClassOf". To explain better let's take the two statements

```
Person rdfs:subClassOf LivingCreature .
Alice rdf:type Person.
```

The first statement uses a special RDFS predicate which has the unique meaning "being a subclass of". Since the meaning is unique the machines are able to manipulate the information according to a certain logic. In our case we design an algorithm that, according to certain rules, is able to infer new information.

The meaning is not context dependent. If the RDFS statements (for example that ones that define subclasses) are exchanged between two different applications they will still keep their meaning because "is subclass of" is a relation that is domain independent. This feature is a step towards the semantic interoperability aimed by the Semantic Web [22].

2.4.1 RDFS reasoning

In the example reported right above we can derive that Alice is a living creature using the relation "subClassOf". This means we can exploit some of the RDFS constructs to derive new information.

Recalling what said in section 2.1, for us doing some reasoning means continuously apply some rules on the input data. In case the input is encoded using the RDFS constructs there are 14 different rules we can use to infer new information [8]. The rules are available on the web [6] but since during this thesis we will often refer to them we report them also in table 2.1.

[6] http://www.w3.org/TR/rdf-mt/#RDFSRules

Number	**If...**	**then...**
1	s p o (where o is a literal)	_ : n rdf:type rdfs:Literal
2	s rdfs:domain x u s y	u rdf:type x
3	p rdfds:range o s p v	v rdf:type o
4a	s p o	s rdf:type rdfs:Resource
4b	s p o	o rdf:type rdfs:Resource
5	p rdfs:subPropertyOf $p1$ $p1$ rdfs:subPropertyOf $p2$	p rdfs:subPropertyOf $p2$
6	p rdf:type rdf:Property	p rdfs:subPropertyOf p
7	s p o p rdfs:subPropertyOf $p1$	s $p1$ o
8	s rdf:type rdfs:subClassOf	s rdfs:subClassOf rdfs:Resource
9	c rdfs:subClassOf $c1$ v rdf:type c	v rdf:type $c1$
10	u rdf:type rdfs:Class	u rdfs:subClassOf u
11	c rdfs:subClassOf $c1$ $c1$ rdfs:subClassOf $c2$	c rdfs:subClassof $c2$
12	s rdf:type rdfs:ContainerMembershipProperty	s rdfs:subPropertyOf rdfs:member
13	s rdf:type rdfs:Datatype	s rdfs:subClassOf rdfs:Literal

Table 2.1: RDFS reasoning rules

During the implementation a particular attention was put in every single rule, trying to optimize the execution of the ruleset in order to speedup the computation.

2.5 OWL

With RDF Schema it is possible to define only relations between the hierarchy of the classes and property, or define the domain and range of these properties. The scientific community needed a language that could be used for more complex ontologies and therefore they started to work on a richer language that would be later released as the OWL language [22].

The language OWL was standardized by W3C on 2001[7]. OWL is a merge of two not-standardized languages DAML [10] and OIL [6]. OWL falls into the category of ontology languages. These are languages that we can use to formally express a particular domain.

OWL is built upon RDF and therefore the two languages share the same syntax. An OWL document can be seen as a RDF document with some specific OWL constructs. However, a complete compatibility between OWL and RDF brings some problems concerning the reasoning and this is due to the high express ability of the RDF primitives.

To fix this the standardization group derived three different versions of OWL that enclose each others. These versions are called OWL Full, OWL DL, OWL Lite [22].

OWL Full corresponds to the full specification of the OWL language. In OWL Full there is a complete compatibility with RDF but this comes at the price of computational intractably. Said in simpler words it is impossible to write a complete and efficient reasoner for OWL Full because this problem is simply undecidable and therefore not implementable by a computer algorithm.

To solve this problem of intractability two smaller subsets of OWL Full were standardized for which it is possible to implement an efficient form of reasoning [27]. These two languages are OWL DL and OWL Lite.

OWL DL and OWL Lite are less expressible than OWL Full and not completely compatible with RDF. However, the big advantage of these smaller languages is that they permit a feasible reasoning. The three languages are one subset of the other.

The main difference between OWL and RDF/RDFS stands on the much higher expressiveness that we can reach with OWL [22]. Two consequences of having an higher expressiveness are that the reasoning is much more sophisticated but also much more difficult to implement in an efficient way.

2.5.1 OWL reasoning

Most of the OWL data present in the web is OWL Full. Even the most common ontologies violate some of the assumptions made in OWL DL [11]. However

[7] http://www.w3.org/TR/owl-features/

reasoning over OWL Full is an undecidable problem and rule-based semantics for OWL Full are (yet) not known [27].

Herman J. ter Horst considered a small fragment of OWL and proposed in [27] a non-standard semantics called the $pD*$ semantics. The fragment $pD*$ has a semantics that is weaker than the standard OWL Full semantics but in $pD*$ the computation of the closure has a low complexity (NP or P in a special case) and can be expressed with a set of rules like RDFS.

These rules are more complex than RDFS because they require multiple joins over the data, or joins between two instance data. Since we will mention the OWL rules many times during this paper, we will report them in table 2.2.

2.6 The MapReduce programming model

MapReduce is a distributed programming model originally designed and implemented by Google for processing and generating large data sets [4].

The model is built over two simple functions, *map* and *reduce*, that are similar to the functions "map" and "reduce" present in functional languages like Lisp [16].

The Google MapReduce programming model has proved to be performant since the simple principle of "mapping and reduce" allows a high degree of parallelism with little costs of overhead. Today the Google's MapReduce framework is used inside Google to process data on the order of petabytes on a network of few thousand of computers.

In this programming model all the information is encoded as tuples of the form *<key,value>*.

The workflow of a MapReduce job is this: first, the *map* function processes the input tuples returning some other intermediary tuple *<key2,value2>*. Then the intermediary tuples are grouped together according to their key. After, each group will be processed by the *reduce* function which will output some new tuples of the form *<key3, value3>*.

2.6.1 Example: Counting occurences of words

Let's take a simple problem that is often used to explain how MapReduce works in practice. The problem consists in counting the occurrences of single words within a text and it can be solved by launching a single MapReduce job.

As first we convert the input text in a sequence of tuples that have both as key and value all the words of the text. For example the sentence "MapReduce is a programming model." will be converted in five tuples each containing one word of the sentence both as key and as value.

Then we define the map algorithm as follows: for every tuple in input of the form *<word,word>*, the algorithm returns an intermediate tuple of the form *<word, 1>*. The key of this tuple is the word itself while 1 is an irrelevant value.

After the map function has processed all the input, the intermediate tuples will be grouped together according to their key. The reduce function will simply

Number	**If...**	**then...**
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 $u\ p\ v$	u rdf:type v
14b	v owl:hasValue w v owl:onProperty p u rdf:type v	$u\ p\ v$
15	v owl:someValuesFrom w v owl:onProperty p $u\ p\ x$ x rdf:type w	u rdf:type v
16	v owl:allValuesFrom w v owl:onProperty p u rdf:type v $u\ p\ x$	x rdf:type w

Table 2.2: ter Horst OWL reasoning rules

Algorithm 1 Example: counting words occurences

```
map(key, value):
//key: word
//value: word
  output.collect(key, 1)

reduce(key, iterator values):
  count = 0
  for value in values
   count = count + 1
  output.collect(key,count)
```

count the number of tuples in the group because the number of tuples reflects the number of times the mappers have encountered that word in the text. It will output the result in a tuple of the form <*word,count*>. This tuple encodes the number of times that the word "word" appears in the input text. An overall picture of this particular job execution is given in figure 2.2 while in Algorithm 1 we report the pseudo code for this example.

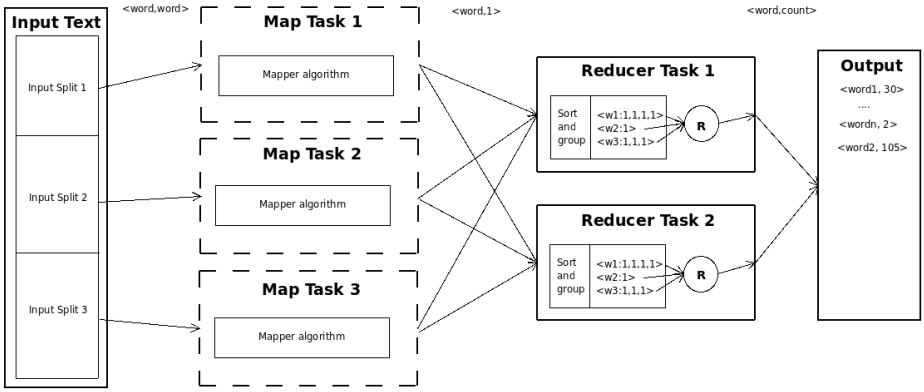

Figure 2.2: Execution of the word count example

2.6.2 Parallelism of the job execution

The map and reduce functions work only with a small fraction of the input and do not need to have access to other data. Therefore the execution of these two functions can be efficiently distributed on several nodes after we have split the input in chunks of equal size. With a MapReduce framework like Hadoop the user can submit some jobs to the framework and the framework will execute them on the available nodes taking care of all the technical details.

A typical execution of a MapReduce job is depicted in figure 2.3. First the master (that in case of Hadoop is called "jobtracker") splits the input in several chunks and then it assigns to the workers the execution of the map tasks on all the input splits. The workers will read the input split that was assigned to them, execute the code of the map algorithm, and store locally their output. The output will be partitioned according to the tuple's key and each partition will be processed by a particular reduce task. The framework will assign each of the reduce tasks to the workers and they will remotely fetch the information they need, execute the reduce algorithm and return the tuples in output.

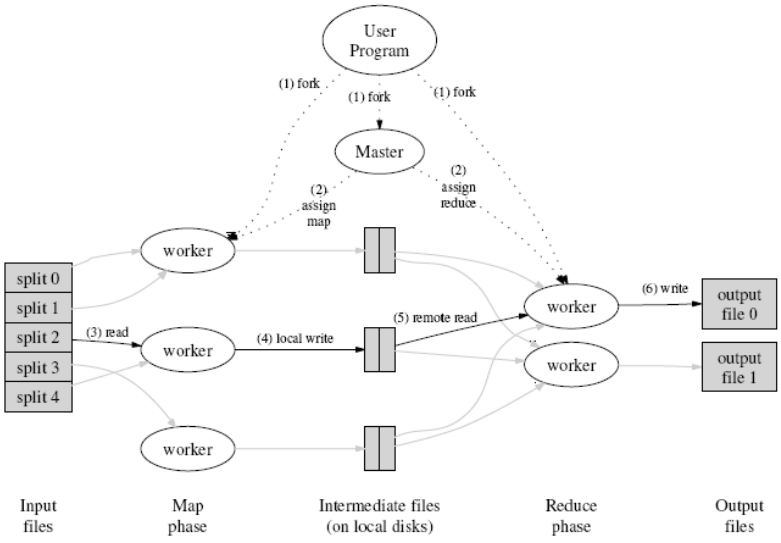

Figure 2.3: Parallel execution of a MapReduce job

2.6.3 Programming rigidity

The simplicity of this paradigm (first map, then reduce) allows the framework to efficiently distribute the several tasks but it also brings a rigidity that does not give so much flexibility in terms of programming. The user cannot do anything else then a sequence of map and a reduce. This is one of the main disadvantages of the MapReduce programming model. Send back the data in output to a mapper or process only particular tuples are two operations that are not possible. Certain operations cannot be encoded with a single map/reduce sequence and the framework is strict on it. The only way to implement complex operations is to launch several consequent jobs. The high performances that we can achieve come at the price of a programming rigidity.

One particular problem that is difficult to treat with the MapReduce framework consists in executing a join between data from different sources. This

operation is difficult because we need to have access to a variety of information at the same time in order infer a relation between it. For example if we want to join the information about a student with the information about the grades we need to have access to both parts at the same time. In MapReduce all the information is encoded as a tuple <*key, value*> and the information contained in one tuple is supposed to be self-contained and independent; in other words MapReduce assumes we do not need information contained in other tuples to process a particular one. This assumption allows the framework to efficiently parallelize the work but in this specific case it is a limitation because, as we have seen in the previous sections, in order to be able to do some reasoning we need to have access to more than one triple at the same time.

2.7 The Hadoop framework

The Hadoop framework[8] is an Java open source project hosted by the Apache Software foundation and it has in Yahoo! one of its biggest contributors. Hadoop implements the MapReduce programming model and it is a free alternative to the Google's MapReduce framework that is not publicly accessible. Actually Hadoop is the most popular and used MapReduce framework and this is mainly due to its open source license and to the intense development that was done in the last years.

The Hadoop framework runs over a network of computers and it uses a distributed filesystem for the data storage. The user is able to launch some MapReduce jobs on the framework which will spread the execution between the nodes in the network. A typical MapReduce job reads the input from the distribute filesystem, processes it and writes the output back on the filesystem. Hadoop offers a distributed filesystem called HDFS that is tightly connected to the rest of the framework. HDFS is not the only choice for the data storage since Hadoop can use other filesystems, like the Amazon S3 for example, but HDFS is heavily integrated with Hadoop and it is easy to install and configure.

The first action taken by Hadoop is to analyze the input text and splits it in some fragments, where each of them will be processed by a single mapper on the network. These fragments are equally divided so that all the mappers (that are executed on different machines) receive the same amount of input.

In the framework there are four different types of programs running. The first is called "jobtracker" and it acts as a sort of master that submits and coordinates the jobs executions on the different computational units. The "namenode" is the counterpart of the "jobtracker" for the distributed system HDFS. The namenode takes care of the replication of the blocks organizing and keeping trace of the nodes activity. The "tasktracker" is the program that actually does the job's computation. A tasktracker accepts the assigned tasks given by the jobtracker and reports to him the results obtained. Every computational node in the network has a tasktracker running on it. The tasktracker can be seen as the worker of the network. The last program is the "datanode" and it is the same of

[8] http://hadoop.apache.org/core/

the "tasktracker" for the HDFS filesystem. An installation of Hadoop contains at least one jobtracker plus many tasktracker and, in case it uses HDFS, one namenode plus many different datanodes. Normally the datanodes are executed on the same machines as the tasktrackers, so that these will contemporary act as a slaves for both the Hadoop computation and for the HDFS storage.

Before the job can be launched it must be properly configured. The framework requires the user to indicate which *mapper* and which *reducer* should be used and how the input and output should be processed.

A MapReduce job works with the information encoded by tuples and the real nature of the data is hidden. It could be that the input consists by single files or by some database tables. A MapReduce job only sees the information encoded as tuples and the real source of the data is unknown. For this reason Hadoop uses some specific types of objects to process the I/O translating the data from their original format to a sequence of tuples. The user must configure the job indicating which types of object should be used for it. There are four types of objects that are involved in this I/O task.

The first is called *input format*. The input format is called at the beginning of the job and its purpose is to divide the input in several input splits, one for every mapper of the job. This object returns for every input split a specific *record reader* that will be used by the mapper to fetch the tuples in input. The record reader reads the content of the input split that was assigned to him and translates the information from a byte oriented view (the content of the file) to a "record oriented" view where the information consists in a list of tuples of the form $<K, V>$. These tuples will be the input of the map algorithm.

There are two other types of objects that are used to write the output of the job. These two are the *output format* and the *record writer*. The first formats the output of the job while the second writes physically the records outputted by the reducer.

Hadoop offers many standard objects to read and write the standard formats. For example Hadoop includes an input format called *FileInputFormat* and a *FileOutputFormat* to read and write the data in some files. If the user needs to preprocess the I/O in a particular way he can implement custom versions of these objects and use them instead of the standard ones.

Chapter 3

Related work

In this chapter we will briefly explain some relevant work done in the area so that the reader can place this thesis in a broader context. First we will report a brief description of some existing reasoners that are available on Internet. This is done in section 3.1. Then we will continue in section 3.2 describing some work about reasoning on a large scale. Finally, in section 3.3, we describe some works concerning the dictionary encoding of the data.

3.1 Classical reasoning

The Java suite openrdf-Sesame [3] offers a RDFS reasoner along with a complete framework for the storage and the manipulation of RDF data in the various formats. Sesame offers also a plug-in mechanism that allows other third part software's to performs different types of reasoning. For example SwiftOWLIM [14] is a plug-in that allows in-memory OWL $pD*$ reasoning. The authors claim that Sesame can load 10-20 million of triples on a machine with appropriate hardware[1].

Another framework written in Java is Jena [18] which provides a rule-based inference engine. The framework provides by default a RDFS reasoner and an incomplete OWL/Lite reasoner but the user can extend the reasoners adding some custom rules. With a persisten layer storage called TDB Jena has loaded 1.7 billion of triples in 36 hours with a single machine[2].

Pellet [25] and FaCT++ [29] are other two programs that do OWL DL reasoning. These reasoners are not typically used for the materialization of new statements but more for checking the ontology consistency or instances classification. We could not find benchmarks with data sets greater than 1 million of instances, therefore it is unclear how they scale with bigger data sets.

The program BigOWLIM [14] (a proprietary version of the above mentioned SwiftOWLIM) can load more than 1 billion triple using sophisticated techniques

[1] http://esw.w3.org/topic/LargeTripleStoreshead-2bd211f37828030e68760a955240be22444d8910
[2] http://esw.w3.org/topic/LargeTripleStoreshead-d27d645a863c4804acbbe268091681ec01ee9903

of disk caching. They also provide OWL reasoning using a single machine and they have reported their performances on [13].

Some benchmark tools were developed to evaluate the performances of the reasoners. Some of these benchmarks consist in tools that generate an artificial data set that can be used by different reasoners. One of them is LUBM [7] which is a program that generates data about universities. We have used it to test the performances of our OWL reasoner. Another similar benchmark tool is UOBM [17], that includes both OWL Lite and OWL DL constructs. There are also benchmarks about querying the data sets with SPARQL: these are BSBM[3] and SP2Bench[24].

3.2 Large-scale reasoning

[11] reports the description of an OWL reasoner which focuses on doing OWL reasoning with data crawled on the web. We borrowed from them the idea of splitting the instance data from the schema data because the last one is much smaller and we are able to load it in memory. They have also implemented the ter Horst fragment but they filter out all the data that could lead to an explosion of the derivation. They motivate the choice of filtering the data because it is possible to find some data in the web that is not standard compliant and just applying blindly the rules results in an enormous amount of derivation which mostly does not make sense. In the article they reported the performances with 100 million triples using 1 machine and the results show sublinear scalability.

Our approach share many concepts with this work, but ours is based on a loosely coupled distributed system while theirs is supposed to use shared data structures to calculate the joins. This makes a direct comparison difficult.

Some approaches to do reasoning with large data sets uses distributed systems to improve the scalability. A good introduction paper about distributed systems is [1]. The authors give a definition of distributed system and describe 15 different programming languages that deal with the parallelism. They also describe what are the differences between sequential programming languages and parallel programming languages. This paper, though is relatively old, is still a good introduction for who does not have experience with distributed systems and programming languages.

An approach for parallel OWL reasoning is reported in [26]. Here the authors propose two solutions where they partition the data in two ways and compute the closure over these partitions on the network nodes. The two approaches differ for the partitioning method. In the first case they partition the data and every nodes compute the closure only on a single partition. In the second case the partition is done over the rules and every single nodes apply only certain rules over all the data. They have implemented it but they report the performances only on small datasets of 1 million of statements.

[21] presents a technique based on a self-organising P2P network where the data is partitioned and exchanged among the different peers. The reported

[3] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html

results are on relatively small datasets and they show sublinear scalability.

[23] proposes an approach where every node executes only certain rules and all the data must go through all the nodes. In this way a single node can become the bottleneck of the system, and this does not scale for large amount of data.

[20] presents a framework built on the top of Hadoop called Pig. Pig offers the possibility to the user to run queries in a SQL fashion over the data translating them in a proper sequence of MapReduce jobs. An interesting user case of Pig is described in [19] where they discuss over the implementation of a SPARQL query engine using this framework.

Another extension of MapReduce is reported in [30] where the authors extends the original programming model made by a map and a reduce adding one last phase called "merge". In this last phase the partitioned and sorted results that are returned by the reducers are merged together in an additional phase. In the paper the authors demonstrate how this model can express relational algebra operators like the joins. Such model could be used to execute a data join between the triples, unfortunately they do not provide any working implementation, therefore we could not test it.

3.3 Dictionary encoding

Dictionary encoding is a general problem that is relevant in many different fields, from information retrieval to data compression.

There are several techniques to store the dictionary in memory. A nice comparison between several data structures is given in [31]. From the comparison it resulted that the fastest data structure is the hash table. In this papers the authors propose also a new hashing algorithm called move-to-front where the accessed value is relocated on the top of the chain. This optimization speeds up the performances of the hash table.

A new data structure, called Burst tries, is proposed in [9] and, though is not as fast as a pure hash table, it outperforms significantly the binary trees.

In case the documents use a large dictionary the amount of main memory cannot be enough to keep it in memory and therefore we need to cache it on disk. There are several implementation available for disk based hashtables, some are offered by libraries like BerkleyDB [4] or by opensource projects like DBH [5].

[4] http://www.oracle.com/technology/products/berkeley-db/je/index.html
[5] http://dbh.sourceforge.net/

Chapter 4

Dictionary encoding as MapReduce functions

We need to compress the triples from the N-Triples format to another one that takes less space. The technique of dictionary encoding is widely used to compress the data and there is much research on the subject.

In this chapter we discuss over several approaches to do so. First we can use Hadoop to distribute the computation and use a centralized Sql server to store the dictionary. This approach is described in section 4.2. Since the centralized Sql server can become the bottleneck of the application, we can instead use a distributed cache to store the dictionary. This second option is described in section 4.3. There is another approach where we only use Hadoop without any other external support. This last approach is faster than the other ones and it is described in section 4.4.

4.1 Why do we need dictionary encoding

The input dataset that we use for our tests consists in a set of about 800 million triples encoded in N-Triples and compressed with Gzip. This dataset contains triples from some of the most common repositories on the web like dbpedia, freebase, wordnet, etc.

The data, compressed with gzip, takes about 8 GB of space. We do not know the size of the uncompressed input but this can be roughly estimated decompressing some files and make a proportion.

We picked up some files as a test. Every single file contains 10000 triples and in a compressed format it takes about 80 KB. The same file, uncompressed, is about 1.4 MB. We can estimate a compression factor of 1:16 thus if the complete dataset in compressed format is about 8 GB then if we decompress it we can expect a size of about 130-150GB.

When we consider whether the data should be further compressed we need to evaluate if it is worth to do so. The tasks of encoding and decoding takes

a time that should be detracted from the gain we obtain working with the encoded data. In other words, if encoding the data brings us an advantage of x in terms of computational speedup, we have to make sure that the time of encoding and decoding the data is not greater than x, otherwise the conversion will be pointless. In our case an obvious conversion consists in replacing the three resources of a triple by three numbers and storing somewhere the table with the association $< number, text >$. Basically what we do with dictionary encoding is to assign an unique number to every resource and represent the triples as the sequence of three numbers. This does not affect the reasoning process because when we reason over two triples we do not care if it is a number or text since the only operation we do with them is a comparison. The encoded text can be later decoded replacing the numbers with the original terms.

In our case the encoding brings a clear advantage: as it will be explained later, during the reasoning phase we need to keep stored in the main memory a small amount of data. In the dataset we used for our experiments this small amount consists in about 2 million triples. If we consider as 25 bytes the average length of an URI then storing a triple will take about 75 bytes. If we multiply this number by 2 millions we conclude that this small amount takes about 140MB, that is an amount still feasible for actual machines but not small considering that there are several Hadoop tasks running on the same machine and that the task itself needs a considerable amount of memory to run. Instead if we substitute the text with numbers, let's say with integers, the triple will occupy only 12 bytes against the original 75. The same data will take only 22MB against the original 140MB.

The main problem in building a dictionary is that we need to have access to a centralized data structure that acts as a dictionary repository. This, of course, is fine when the input size is not so big and the job can be done by a single machine, so that the dictionary can be kept in memory. This is not our case because the data is too big. Consider that in our input there are about 300 million unique resources. Recalling our previous estimation of the size of the URIs, a single row of the table *<number, text>* takes about 84 bytes, so if we multiply it for 300 millions it will take in toto 24GB, that is a prohibitive amount for a single standard machine.

4.2 Dictionary encoding using MapReduce and a central database

A first approach could be to write an Hadoop job that reads the triples in textual form, contacts a centralized Mysql server, fetches the associated number for the text and return the triples converted. The advantage in doing so is that we can parallelize the conversion of the triples since the execution is distributed on many nodes but the problem stands on the fact that all of these nodes need to communicate with a centralized server. We can make an estimation of how long this communication process takes. Let's assume that a single query to a

Mysql database can take 10ms (counting the latency of the communication over the network and the processing time on the server side). We have 800 millions triples, therefore we need to make 2.4 billion queries, one for each resource. If we multiply this number for 10 ms we see that it will take approximately 6.5 million of hours. If we parallelize the computation splitting the task over the nodes in Hadoop, we need about 6.5 million computers to finish the task within 1 hour, assuming that the server can serve millions of requests at the same time. We see that even if we reduce the querying time to 1ms from the initial 10ms the approach is clearly unfeasible.

4.3 Dictionary encoding using MapReduce and a distributed hashtable

To solve the problem of the centralized server's bottleneck we can analyze the frequency of the unique resources in the input dataset to see if there are some URIs that are more popular than others. If the popular URIs outnumber significantly the others we can use a cache to keep the most popular in memory and access to the database only if the URI is not present. For this purpose we can implement two simple Hadoop jobs that count the unique resources in the data and report in output their frequency's distribution.

These two jobs are very simple: the first reads in input the triples and during the mapper phase it outputs the three parts of the triple as *<Resource, 1>*. In the reduce phase the key is filtered (because the duplicated URIs are grouped together) and the frequency is counted in a similar way than in the word count example explained in section 2.6. The second job reads in input the output of the previous job and counts how many URIs have the same frequency so that we can draw a distribution of the frequency.

From the analysis of the distribution we can evaluate whether a cache will help or not. In case it does we can use an hash table to store the dictionary instead of a relational table. A distributed hash table distributes the information of the hash table among different nodes caching locally the most popular ones. The advantage of this approach is that we are not using a centralized structure anymore and that we can exploit that advantages of caching on every node the most popular terms. There are several implementation of distributed hash tables, but for this work we only have considered HBase.

HBase is a subproject of Hadoop and therefore it is well integrated with the rest of the framework. It is a relatively young project but it is under heavy developed and it has the support of big companies like Yahoo!. With HBase we can store and retrieve elements from hash tables but we cannot do any relational operations like joins or projections. The hashtables in HBase are column-driven and are meant to store billion of entries. HBase is used nowadays to store tables of billion of rows each with million of columns.

In our case we can use an hash table in HBase as a data structure to store the dictionary and the nodes can access it when they need to encode or decode the

data. However there is a problem of how to assign the number to the resources. If we use a centralized Mysql server we can set the "number" column as auto incremental and every time we insert a new term in the table the database can assign automatically a number to it. In HBase this is not possible and we have to first manually assign a number to a resource and then upload the tuple into the hashtable. In our tests this approach did not perform well, and since the queries to the distributed cache were slower than 10ms it was soon abandoned.

4.4 Dictionary encoding using only MapReduce

Here we show how we can do dictionary encoding using only MapReduce functions without any other external support. We do this by using two MapReduce jobs that first assign to every resource an unique number and then rebuild the triples using the numbers instead of the text.

4.4.1 Overview

What we do is to exploit the ability of Hadoop to partition and group the keys during the reduce phase to assign the numbers to resources.

The first job reads the triples in input, during the map phase it assigns to every triple an unique id and it outputs three tuples with as key the three resources the compose the triple and as values the triple id and the resource's position in that triple. The reducer does two things. First it assigns to the resources an unique id. Then it iterates over the values of the tuples returning for every value a new tuple with the number associated to the resource as key and the triple id in which the resource appeared plus its position in it as value.

The second job reads in input the output of the previous one. The mapper swaps the key with the value returning a tuple that has as key the triple id and as value the resource itself plus the resource's position in the triple. The reducer will group together the tuples that have the same triple id. Every group will have three tuples, each of them having as value the resource number plus the resource position in that triple. With this information the reducer is able to return in output the original triple with the resource numbers instead of the resource text.

In the next two sections we will present these two jobs more in detail.

4.4.2 First job: assign number to the URIs

The first job receives in input in compressed files that contain the triples in the N-Triples format.

For this particular task we have developed a custom input format and a record reader so that the input could be automatically uncompressed and returned as a tuples of the form <*Null, Text*> where Text is a single triple coded in N-Triples format. It is nice to remark again how the Hadoop framework allows the user to abstract completely the underlying nature of the input. The

input could be made of simple files or consists in a database table. The record reader hides completely the nature of the data simply returning a sequence of tuples <*key,value*>.

The mapper first assigns to every triple in input an unique id. Since the mappers are executed on different machines and one triples is processed only by one mapper, we must take care that every mapper assigns an unique id to every triple. What we can do is to partition the number space in such a way that every mapper task can only assign numbers within a certain range. For example the first mapper task can only assign numbers from 0 to 100, the second mapper task only from 101 to 200 and so on. In this way we avoid the risk that two mappers assign the same number to two different triples.

After, the mapper splits the triples in input in three parts: subject, predicate and object. It outputs three different tuples that have as key each of these three parts. The tuples values consist in a number that encodes the triple id (that is the same for all the three) and the position of the resource in the triple.

The encoding of the tuple's value is done in the following way. In the first 7 bytes it encodes the triple id. The triple id is composed as follows. In the first 3 bytes the task encodes its own task id. The task id acts as a sort of marker that indicates the starting point of the range in which the mapper can assign the numbers. In the next 4 bytes it writes the value of a local counter that the mapper increases for every triple in input. The combination of the task id and of the local counter is an unique number that represents the triple's id.

In the last byte of the tuple's value we encode the position of the resource in that particular triple.

For example if we have in input the triple:

```
<s> <p> <o> .
```

the mapper will output the three tuples

```
<s, triple_id+subject>
<p, triple_id+predicate>
<o, triple_id+object>
```

The resources will be grouped together during the reduce phase. First the reducer assigns an unique number to each of the resources. Then it iterates over the values associated with the key and outputs a tuple having as key the number assigned to the resource and as value the one fetched from the iterator.

The ids are assigned in a similar way than before, with the task id on the first 3 bytes and a local counter value in the last 4. With this approach we won't assign consequent numbers because it can be that the number of input processed by the reducer is smaller than the numbers available in the range. However this is not a big issue because: first, we are more interested in compressing the output than in building a nice and efficient dictionary and second, we can limitate the gap between the partitions choosing a range that is as close as possible to the real size of the reducer's input so that there are only few numbers left out. The algorithm is reported in Algorithm 2.

Algorithm 2 Dictionary encoding: first job

```
map(key, value):
 //key: irrelevant
 //value: triple in N-triples format
 local_counter = local_counter + 1
 triple_id = task_id in first 3 bytes + local_counter
 resources[] = split(value)
 output.collect(resource[0], triple_id + subject)
 output.collect(resource[1], triple_id + predicate)
 output.collect(resource[2], triple_id + object)

reduce(key, iterator values):
 //I assign an unique id to the resource in 'key'
 local_counter = local_counter + 1
 resource_id = task_id in first 3 bytes + local_counter

 for (value in values)
  output(resource_id, itr next)
```

The tuples in output contains the relations between the resources and the triples. The only moment when we have access to the resource both as number and as text is during the execution of the reducer task, therefore here we must take care of building the dictionary that contains the association <*number,text*>. The dictionary can be stored in a distributed hash table, in a mysql database or simply in some separated files. If we do not store the dictionary we won't be able to decode the triples to their original format.

4.4.3 Second job: rewrite the triples

This second job is easier and faster to execute than the previous one. The mapper reads the output of the previous job and swap the key with the value and it sets as the tuple's key the triple id and as value the resource id plus the resource position.

In the reduce phase the triples will be grouped together because the triple id is the tuple's key. The reducer simply scrolls through the values and rebuild the triples using the information contained in the resource id and position.

In our implementation the triples are stored in a Java object that contains the subject, the predicate and the object represented by long numbers. The object takes 24 bytes of space to store the resources plus one byte to record if the object is a literal or not. That means that one triple can be stored on disk with only 25 bytes. There are 800 million of triples, therefore the uncompressed encoded version requires only 19GB to be stored, against the 150GB necessary

Algorithm 3 Dictionary encoding: second job

```
map(key, value):
  //key: resource id
  //value: triple id + position
  output.collect(value.triple_id, key + value.position)

reduce(key, iterator values):
  for(value in values)
    if value.position = subject do
      subject = value.resource
    if value.position = predicate do
      predicate = value.resource
    if position = object_not_literal do
      object = value.resource
      object_literal = false
    if position = object_literal do
      object = value.resource
      object_literal = true
    output(null,
           triple(subject, predicate, object, object_literal))
```

to store the triples in the original format. Using this technique the compression level is about 1:8.

4.4.4 Using a cache to prevent load balancing

The first job can suffer of a load balancing problem that can slow down the computation. If a particular resource is very popular there will be many intermediate tuples with the same key and all of them will be sent and processed by a single reducer task that is executed on a single machine.

This load balancing problem is limited by the fact that we send only the triple id and not the complete triple as value, so that even the reducer has to process more values these are just 8 bytes each, so that the difference is still little. If the frequency of the nodes is spread uniformly the difference is not noticeable but if there are few extremely popular resource then the problem arises also if we just use numbers. To solve this problem we can initially identify which resources are extremely popular and prevent by being processed by a single reducer task.

We can identify the popular resources by launching a job that counts the occurrences of the resources in the data. After we have done it we can, manually or with a script, assign a number to the most popular resources and store this association in a small text file.

This file can be loaded by the mapper and kept in memory in a hash table as

a in-memory cache. Before the mappers output the tuples they check whether the resources are present in this hash table. If they are, instead of setting the resources as key they set a specific fake key of the form:

```
"@FAKE" + random_data + "-" + ID_MANUALLY_ASSIGNED
```

The random data prevents that all the tuples with the same ID end in the same reducer. When the reducer assigns a number to the key it checks first if the key starts with "@FAKE" and if it does it simply extracts the end of the string and use that number instead of a new one. In this way we assure that all those resources will get the same number even if they are processed by different reducers. Using the cache we greatly decrease the impact of load balancing with the effect of speed up the computation.

Chapter 5

RDFS reasoning as MapReduce functions

In section 2.4.1 we described the theory behind RDFS reasoning. We defined it as a process in which we keep applying certain rules to the input set until we reach a fix point.

The RDFS reasoning can be done in different ways. The development process of the algorithm started with a very simple and dummy implementation and it went through many changes until it ended in a final and tested version.

This section describes three possible algorithms starting from the first naive one and ending in a more refined and optimized version. This explanation reflects the development process of the program that was developed along with this work. The first two versions are no longer present in the current code because they are obsolete and inefficients. The third and final algorithm is described more in detail and it is the one used to evaluate the performances of our approach.

5.1 Excluding uninteresting rules

The RDFS reasoner consists in a ruleset of 14 rules (reported in table 2.1) that we have to apply to the dataset. In this ruleset there are certain rules that are trivial and uninteresting and we will exclude them from our discussion. Those rules are rule number 1,4a,4b,6 and 10. The decision of leaving them out lies on the fact that they produce a trivial output that cannot be used for further derivations. Said in other words, they do not produce information that is usable for our reasoning purposes. These rules work using only one triple in input and, in case the user requires the reasoner to behave as a standard RDFS reasoner (therefore returning also this trivial output), we can easily implemented by simply apply a single and load balanced MapReduce job at the end of the computation.

To explain better why these rules are unimportant let's take for example the

rules 4a and 4b. Basically what these rules do is to explicitly mark any URI that is subject or object in a triple as a type of *rdfs:Resource*. These rules are trivial and indeed very easy to implement, but the output of them cannot be used for further derivation. The only rules that could fire using the output of rules 4a and 4b are rules 2,3,7 and 9 but they will never do. The reason stands on the fact that if we want that rules 2 and 3 fire then *rdf:type* must have a domain or a range associated with it. Also, if we want that rules 7 and 9 fire then *rdf:type* must be defined as a subproperty of something else or *rdfs:Resource* as a subclass of something else. Since both *rdf:type* and *rdfs:Resource* are standard URIs defined by the RDF language the user is not allowed to change their semantic adding custom domains or ranges or defining them as subclass of something else. If there are such triples in input they should be simply ignored because these are dangerous triples that try to redefine the standard language (more about ontology hijacking is said in section 3). If we assume the input data is clean and comply to the standard then rules 2,3,7 and 9 will never fire and we can safetly claim that rules 4a and 4b produce an output that cannot be used for further reasoning. The same motivation applies also for the other excluded rules. After we leave these uninteresting rules out, the ruleset becomes a subset made of 9 different rules.

5.2 Initial and naive implementation

In the program that we developed we first apply a technical job that converts the data in N-Triples format in a more convenient format for us. This job is not important for our reasoning task because it is strictly technical. The only thing it does is to convert the information in Java objects stored in files called "SequenceFile".

A "SequenceFile" is a file that stores the information as tuple of $<key,value>$. Both the keys and values are particular Java objects that implement a specific Hadoop interface called "WritableComparable". This interface defines some methods that the objects use to serialize themselves in a stream (that is a filestream in case of the SequenceFile). The advantage of storing the input data using a sequence file lies on the fact that Hadoop is able to automatically compress and decompress these files and it offers predefined input readers that read them in a transparent way. The information is already encoded as $<K,V>$ so the input can be read in a straightforward way without any conversion.

In this initial version the reasoning is done by 9 different jobs, where each of them implements one rule. These jobs are repetitively launched until we reach a fix point. We will not describe all the single jobs because they are all very similar to each others. Instead we will just pick one rule and report a job as example.

5.2.1 Example reasoning job

The rule we take as example is:

```
if   a rdf:type B
and  B rfds:subClassOf C
then a rdf:type C
```

If we want to apply this rule we first need to search if there is a common element between any two triples in the input data. The triples must comply with the format written in the rule's antecedents. In this particular example one triple must have as predicate the URI *rdf:type* while the second must have instead *rdfs:subClassOf*. The two triples must share at least one element, in this particular case the object of the first must be the same as the subject of the second.

We can describe the reasoning as a process that first try to join different triples, and then, in case the join is possible, asserts of new information. If we see the derivation process under this point of view, we can rewrite the problem of applying a particular rule as the problem of finding all the possible joins between the information in the dataset.

We can implement a data join for the rule we reported above with one MapReduce job. In such job the mapper checks whether the input triple is a "type" triple or a "subclass" triple. If it is a type triple it outputs a new tuple with as key the object of the triple and as value the triple itself. Instead, if it is a subclass triple, it outputs as key the triple's subject instead of the object.

The reducers will group the tuples according to their key. That means that if there are some "type" triples with the object equals to the subject of other "subclass" triples they will be grouped together. The reducer will iterate the list saving in memory the two types of triples in the group. It will then proceeds emitting the new derived information. We report the algorithm in Algorithm 4.

All the nine different rules have the same structure: the mappers output the tuples using as keys the possible URIs that could be the match points. The reducers will check whether the match occurs and when it happens they will simply output the new information.

There are many problems in this first implementation. The first and most important is a load balancing problem. If an URI is very popular and used in many joins then all the triples involved with that URI will be sent to a single reducer that must be able to store them in memory. Another problem relies on the fact that we need to launch a variable number of jobs depending on the input data. For example, if we take the subclass transitivity rule (the 11th rule in table 2.1), it will take $logn$ jobs to derive everything if the input contains a subclass chain of n triples (a subclass b, b subclass c, c subclass d and so on).

All of this makes the implementation unusable and indeed this version was abandoned almost immediately. The second implementation approaches the problem in a different way to overcome the two problems just illustrated above.

5.3 Second implementation

The main problems that were encountered for the previous implementation can be resumed in the list:

Algorithm 4 RDFS reasoning: example naive implementation

```
map(key, value):
 //key: irrelevant
 //value: triple
   if value.predicate = rdf:type then
    output(value.object, value)
 else if value.predicate = rdfs:subclass then
    output(value.subject, value)

reduce(key, iterator values):
  for triple in values
    if (triple.predicate = rdf:type then
      types.add(triple)
    else
      subclasses.add(triple)

  for (typeTriple in types) do
    for (subclassTriple in subclasses) do
      newTriple = typeTriple.subject, rdf:type, subclassTriple.object
      output(null, newTriple)
```

- load balancing problem: doing a join in the reducer phase can be problematic in case there are very popular URIs shared by many triples;
- complexity program: if there is even one single chain of subclasses in the data we need many more jobs to compute a complete closure.

The second implementation solves these two problems rearranging the rule's execution order and loading the schema triples in memory.

5.3.1 Rule's execution order

We notice that all the nine rules output four different types of triple. Rules 5 and 12 output schema triples that define subproperty relations. All these triples can be distinguished by the others because they have *rdfs:subPropertyOf* as predicate. Rules 8, 11 and 13 return similar triples than the previous rules but with *rdfs:subClassOf* as predicate. Rules 2, 3 and 9 output data triples that define the type of some particular resources. All these triples share *rdf:type* as predicate. Only rule 7 outputs a triple that can be of every nature.

If we look at the antecedents of the rules we are also able to make certain divisions. Rules 5 and 10 only work with schema triples that either define subclass of subproperty relations between the resources. Rules 8, 9, 12 and 13 work with a subset of the triples with (*rdf:type*), (*rdfs:subClassOf*) or (*rdfs:subPropertyOf*)

as predicate. Only rule 2, 3 and 7 can virtually accepts any triple in input. The relations between the rules are useful because if we launch the rules execution in a special order we can limitate the number of jobs that are necessary for the complete closure. Figure 5.1 reports the connection between these rules.

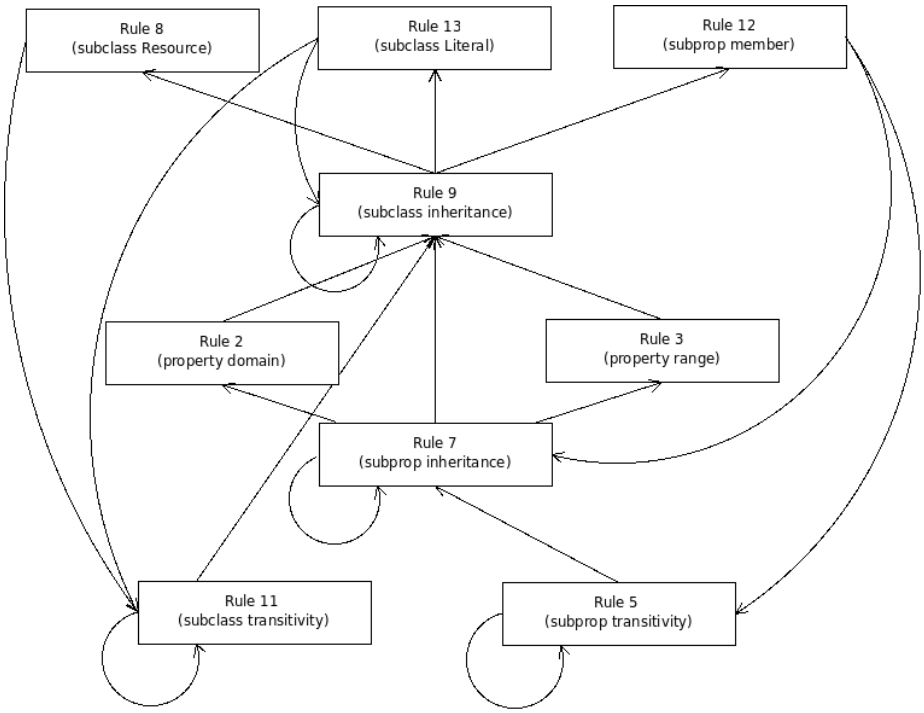

Figure 5.1: RDFS rules relations

The ideal rule's execution should start from the bottom of the picture and go to the top. That means we should first apply the transitivity rules (rule 5 and 11), then apply rule 7, then rule 2 and 3, then rule 9 and finally rules 8, 12 and 13. These three rules potentially can output triples that can be used by the initial transitivity rules but if we look more carefully we see than this is not the case. Rule 8 outputs a triple that could be used by rules 9 and 11. We exclude the case of rule 9 because the output would be that a resource x is type *rdfs:Resource*. This statement is valid for every resource and it can be derived also through rule 4a that is one rule that was excluded because trivial. Rule 8 could make rule 11 fire. We exclude the case that there is a superclass of *rdfs:Resource* because we assume that the schema may not be redefined, but it could be that there is a subclass of it. In this case we will materialize a statement that says that something is a subclass of *rdfs:Resource*. This statement cannot be used for any further derivation, therefore the cycle ends with it.

Rule 12 outputs a triple of the form *s rdfs:subPropertyOf rdfs:member*. This could make rules 5 and 7 to fire. In both cases the output would be a triple that

cannot be used anymore. The same case is with rule 13. Rules 9 and 11 could fire but they would output either triples of the form *s rdfs:subClassOf rdfs:Literal* or *s rdfs:type rdfs:Literal* (the correctness of this last one is debatable since we are assuming that a resource is a Literal). Anyway in both cases these triples will not lead to any derivation.

The most important consideration is that these last rules cannot generates a derivation that can make these rules firing again. In other words there is no main cycle in the rules execution. This consideration is important because we it means do not need to relaunch the same job more than once (except in the case the rule is recursive with itself). After we have applied rules 8, 12 and 13 we have reached a fixed point in our closure computation, because even we can derive some statements, those will not lead to any new derivation. In case we need to compute a complete RDFS closure, we can further process these last 3 rules, deriving the last statements and apply the trivial rules that we have excluded at the beginning of our discussion. This last part can be easily implemented, and therefore we will exclude it from our discussion.

The only loops are the ones between the rules themselves. For example, if we want to apply rule 11, we need to apply it repetitively until we cannot derive anything anymore. Only after it we can safely continue with our process. There are only 4 rules (rule 2, 3, 8, 12 and 13) that are not recursive. These inner loops are problematic because the numbers of repeats depends on the input data and we want to reduce the number of jobs to the minimum necessary. We solve this problem by loading some schema triples in memory.

5.3.2 Loading the schema triples in memory

We exploit the fact that the number of schema triples is much lower than the data triples and that all the joins we do are between a big subset containing the instance triples and a much smaller one containing the schema triples. For our rules we need four subsets of the schema triples. The first is the one that defines the domain of properties. This subset is used in rule 2. The second is the subset that defines the range of the properties. This is used in rule 3. The third is the one that defines the subproperty relation and it is needed in rules 5 and 7. The fourth and last is the set with the subclass relations used for rules 9 and 11.

Since these four datasets are small we can store them in memory, and instead of doing the join using a single job as we did before we check if there is a match between the input triples with the triples that are kept in memory. For example if we want to apply the rule 9 we keep in memory all the triples that regard the subclass relations and, for every triple of the form *a rdf:type B*, we check if the triple's object matches with one or more of the triples we keep in memory. If it does then we derive a new statement.

We report a brief example to explain it better. Consider the set of input triples:

```
a1 rdf:type A .
```

```
b1 rdf:type B .
A rdfs:subclassof B .
B rdfs:subclassof C .
C rdfs:subclassof D .
```

The nodes load the "subclass" triples in a in-memory hash table with the subject of the triples as key and the object as value. Following the example this hash-map will contain the entries:

key	**value**
A	B
B	C
C	D

When the mapper receives in input the triple *a rdf:type A* it checks whether there is a key equals to the triple's object in the hash table. In this case the mapper succeed and the mapper fetches the value associated with A. This value is B and the mapper outputs the new triple *a rdf:type B*.

The advantage of keeping the schema triples in memory is that we do not need to launch the job more than one time. In our previous approach we needed to launch the same job again and again until we reach a fixed point. If we keep the schema triples in memory we can check if there are matches in a recursive way. For example if we take rule 9 the output produced by applying the rule can be further used as input to check whether it could lead to a new derivation with the schema in memory. In this way we do not need to relaunch another job.

After this last consideration we eliminate also the inner loops and our reasoning process becomes a linear sequence of jobs.

In this implementation the join is done during the map phase. The reducers simply filter the derived triples against the input so that we avoid to write in output the same triple more than one time. The eight rules (rule 8 is not considered in this version) are implemented in two Hadoop jobs. The first job executes the two transitivity rules. The second applies all the other rules. In the next two subsections we describe these two jobs more in detail.

5.3.3 First job - apply transitivity rules

First we apply rules 5 and 11. These rules exploit the transitivity property of the subclass and subproperty relations making explicit every subproperty and subclass statement. The algorithm is reported in Algorithm 5.

The mapper checks if the triple in input matches with the schema. In case it does, it simply emits the new derived triple, setting it as the tuple's key, and setting *true* as value. It also outputs the input triple, with the only difference that it sets the value as *false*.

The reducer code is very simple. It checks whether the triples in input are derived or not. It does that iterating over the values of the tuples. If there is a value set to false it means that the triple was present in input. In that case

Algorithm 5 RDFS reasoner: second version first job

```
map(key, value):
 //key: irrelevant
 //value: triple
 if value.predicate = rdfs:subClassOf then
   objects = subclass_schema.recursive_get(value.object)
   for object in objects
    output(triple(value.subject, rdfs:subClassOf, object),true)

 if value.predicate = rdfs:subPropertyOf then
   super_objects = subprop_schema.recursive_get(value.object)
   for (object in super_objects)
     output(triple(value.subject, rdfs:subPropertyOf, object), true)
 output(value,false)

reduce(key, iterator values):
 for value in values
   if not value then
     exit
 output(null,key)
```

the triple is not outputted. Otherwise it is outputted only once. The reducer assures us we derive only unique triples.

5.3.4 Second job

The second job implements the remaining rules. In Hadoop there is a special mapper called *ChainMapper* that we use to chain some mappers one after the other. The output of the previous mapper becomes the input of the following, and so on until the last mapper outputs the data to the reducers.

We define 3 different mappers. The first implements rule 7. It loads in memory the sub-property triples and it checks whether the predicate of the input triples is contained in the schema. This rule is the first executed in the chain. The output of this mapper goes to the second mapper that encodes rules 2 and 3. It loads in two different data structures the schema triples that concern the domain and range of properties. These two rules are grouped together in one mapper because they are independent from each others and therefore they can be applied at the same time. To notice is that the choice of putting first the mapper that implements rule 7 before this last one is not casual but due to the fact that the output of rule 7 can be used by rules 2 and 3. In this way the output of rule 7 is checked in the second mapper ensuring we do not miss any new derivation.

The last mapper contains the implementation of rules 9, 11, 12 and 13. These four rules are all grouped in one mapper because rules 9 and 11 use the same memory data structure while rules 12 and 13 simply needs triples that either are in input or just outputted from the previous rules. The algorithm is reported in Algorithm 6.

The mapper first checks if the input triple meets certain conditions and when it does, it checks recursively if there is a match with the memory data structure. The triple produced is then forwarded to the reducer that will filter out the duplicates.

This implementation was tested on the input dataset but there was a main problem that concerned the number of duplicates generated during the map phase. In one of the tests the number of duplicates grew so much until it consumed all the space offered by our Hadoop cluster. The reason stands on the fact that we process triple by triple even if they both could lead to the same derivation. Let's take an example. There are these 6 triples:

```
a rdf:type C
a rdf:type D
C rdfs:subclass E
D rdfs:subclass E
E rdfs:subclass F
F rdfs:subclass G
```

The mappers will load in memory the last 4 triples. When the mapper receives the first triple it will derive that a is a type of E, F, G. When it receives the second triple it will also derive the same 4 triples. If there are long chains of subclass triples there will be an explosion of duplicated triples, and though all of them will be later correctly filtered by the reducer, they must be first locally stored and later sent to the reducer.

The third and final implementation faces this last problem and proposes a solution that turns out being a much more performing algorithm than the ones presented so far.

5.4 Third and final implementation

The previous implementation generated too many duplicates and resulted in being unfeasible. Here we present an improved and final version of the algorithm that was successfully implemented and used to evaluate the performances of our approach. First we have moved the join's execution from the map to the reduce phase. This choice can generate some duplicates but allows us to reduce the number of duplicates. We have also slightly rearranged the rule's execution order in five different jobs, but still in a way that we do not need to relaunch the same job more than once.

The first job executes rules 5 and 7. The second one executes rules 2 and 3. The thirds job cleans up some duplicates that could have been generated in the previous jobs and the fourth applies rules 8, 9, 11, 12 and 13. The fifth and

Algorithm 6 RDFS reasoner: second version second job

```
/**** CODE FIRST MAPPER ****/
map(key, value):
 super_properties = subprop_schema.recursive_get(value.predicate)
 for property in super_properties
   output.collect(triple(value.subject, property, value.object))
 output(value, false)
/**** CODE SECOND MAPPER ****/
map(key, value) {
 if (domain_schema_triples.contains(value.predicate))
   domains = domain_schema.get(values.predicate)
   for domain in domains
     output.collect(triple(value.subject,rdf:type,domain),true)
 if (range_schema_triples.contains(value.predicate))
   ranges = range_schema.get(values.predicate)
   for range in ranges
     output.collect(triple(value.object,rdf:type,range),true)
 output(value, false)
/**** CODE THIRD MAPPER ****/
map(key, value) {

if triple.predicate = rdfs:subClassOf then
  super_classes = subclass_schema.recursive_get(value.object)
  for class in super_classes
    output.collect(
           triple(value.subject,rdfs:subClassOf,class),true)
if triple.predicate = rdf:type then
  super_classes = subclass_schema.recursive_get(value.object)
  for class in super_classes
    output.collect(
           triple(value.subject,rdf:type,class),true)
    if class = rdfs:ContainerMembershipProperty then
      output.collect(
      triple(value.subject,rdfs:subPropertyOf,rdfs:member),true)
    if class = rdfs:Datatype then
      output.collect(
        triple(value.subject,rdfs:subClassOf,rdfs:Literal),true)
   //Check special rules also on the input values
  if class = rdfs:ContainerMembershipProperty then
    output.collect(
     triple(value.subject,rdfs:subPropertyOf,rdfs:member),true)
  if class = rdfs:Datatype then
    output.collect(
       triple(value.subject,rdfs:subClassOf,rdfs:Literal),true)
output.collect(value,false)
/**** REDUCER ****/
reduce(key, iterator values):
 for value in values
   if not value then
     exit
 output(null,key)
```

last job processes the output of the rules 8, 12 and 13 to derive the last bit of information. This job derives information that could be considered trivial, like being subclass of Resource, therefore, in case the user does not need to have a complete reasoning, it can be left out.

In Figure 5.2 we report a sketch of the overall jobs execution. We see from the picture that first the data is encoded and then the five jobs are executed right after it. The output of the jobs is added to the input so that every jobs reads in input the original data plus everything that was already derived. In the next subsections we will first discuss more in detail about the problem of the duplicates, explaining why we moved the join execution from the map to the reduce, and then we will continue describing all the single jobs.

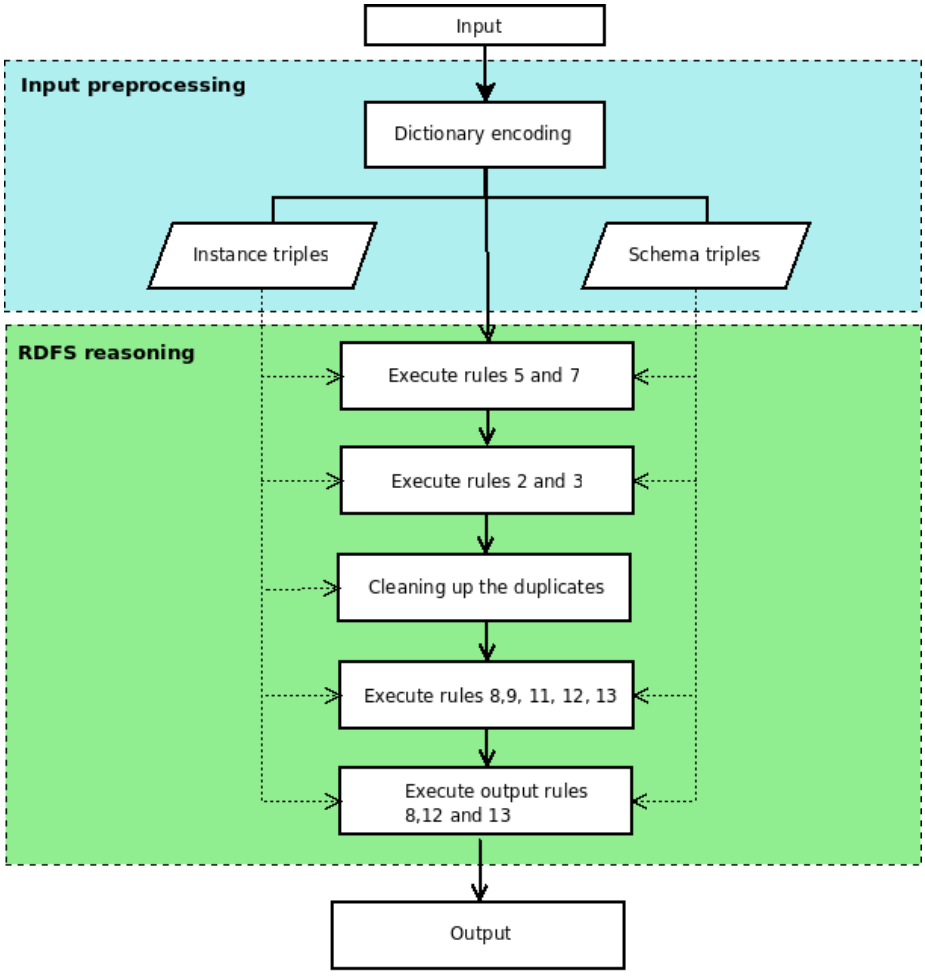

Figure 5.2: Execution of the RDFS reasoner

5.4.1 Generation of duplicates

The problem of generating an high number of duplicates which the previous implementation suffered of can be solved by grouping first the triples that can potentially generate the same output and then applying the rules only one time per group. For this reason we use first the map phase to group the triples and then we execute the join on the reducer phase.

We must group the triples in a particular way. Every derived statement contains a part that comes from the input triples and another part that comes from the schema. To explain better this concept let's take a simple rule, rule 7, as example. In this rule the subject and the object of the derived triples come from the input while the predicate comes from the schema.

In rule 7 the duplicated triples are derived from the input triples which share the same subject and object and have different predicates that lead to the same derivation. If we process these triples singularly, as we did before, they both derive the same statements because they are not aware of the existence of the others. If we want to avoid the duplicates we need to first group these triples together and then process them only one time.

If we group the triples using the part of them that is also used in the derivation and then we execute the join only one time per group, we will never produce some derivatives, because every group has a different key.

During the map phase we set the parts of the input triples that are also used in the derived triples as the tuple keys and the parts that should match the schema as values. For example for rule 7 we will put as key the subject and the object of the input triple, for rule 5 only the subject, and so on. If we add more elements in the key (for example putting also the predicate in case of rule 7) this condition won't hold anymore. If we put less elements we will still avoid duplicates but our groups will be bigger introducing some load balancing problems.

During the reduce phase we iterate over the possible matching points and we calculate the derivation set they produce, filtering in memory eventual duplicates. After, we simply output the derived statements using the fixed part that comes from the group's key and the elements derived from the schema triples.

With this procedure we do eliminate the duplicates between the derived triples but when we do not delete the duplicates between the input triples and the derived ones, unless we forward all the input triples to the reducers. That means that after the reasoning jobs we need to launch another job only to clean up these duplicates.

5.4.2 First job: subproperty inheritance

The first job implements rule 7 and 5.

The mapper receives in input one triple at the time. If it is a subproperty triple we can apply rule 5 and check whether the object matches in the schema. When it does the mapper will output a tuple settings as key a special flag to identify that this triple should be processed by rule 5 plus the triple's subject.

As value the mapper sets the object of the triple.

Otherwise, if it is a generic triple, the mapper checks whether the predicate matches with the schema. If it does it outputs a tuple setting as key a different flag plus the triple's subject and object. The mapper sets as value the triple's predicate.

The framework will group the tuples by the key and the reducers will process each of these group one by one. The first thing a reducer does is to check the flag. According to the flag it processes the group following a specific logic (that can be rule 5 execution or rule 7 execution).

In both cases the procedure is similar. First we collect the values storing them on a hashset. This operation eliminates eventual duplicates in case the input contains duplicated triples. After, for each value, the reducer recursively calculates all the superproperties and it stores them in a hash-set so that we also eliminate the duplicates within the derivation output. After these two operations the reducer continues emitting the new triples: in case we apply the rule 7 we extract the subject and the object from the key and derive the new triples using the super-properties as predicate. In case we apply the rule 5 we will use the key as subject and the super-properties as object (the predicate will be *rdfs:subPropertyOf*).

The algorithm is reported in Algorithm 7.

5.4.3 Second job: domain and range of properties

This job is similar to the previous. Here we apply rules 2 and 3. The mapper checks in a similar way than before whether the predicate has a domain or a range associated. Here we do not divide the two rules in two groups setting a flag in the key but instead we process them as they are a unique rule. We do this because otherwise the output of one rule can generates duplicates with the other ones. Consider as example the fragment:

```
a p b .
c p2 a .
p rdfs:domain Thing .
p2 rdfs:range Thing .
```

If we apply rules 2 and 3 separately they will both derive that a is type $Thing$. To avoid that we need to group the triples together. In case the predicate has a domain we output a tuple that has as key the triple's subject and as value the predicate plus a special flag. Otherwise, if the predicate has a range, we output the triple's object as key and as value the predicate with another flag. The flag in the value is needed because we need to know which schema we should match the predicate against. The derivation output is put in a hash-set so that we eliminates the duplicates and we output the triples using the key as subject and the derived output as object.

The algorithm is reported in Algorithm 8.

Algorithm 7 RDFS reasoning: third version, first job

```
map(key, value):
 //key: irrelevant
 //value: triple
  if (value.predicate = rdfs:subPropertyOf
      and subprop_schema_triples.contains(value.object)) then
   //We can apply rule 5
   key = '0' + value.subject
   output(key, value.object)

 if (subprop_schema_triples.contains(value.predicate) then
  //We can apply rule 7
  key = '1' + value.subject + value.object
  output(key,value.predicate)

reduce(key, iterator values):
 //valuesToMatch is a hashset
 for (value in values)
   valuesToMatch.add(values)

 //Superproperties is a hashset
 for(valueToMatch in valuesToMatch)
   superproperties.add(schema_triples.recursive_get(valueToMatch))

 if (key[0] == 0) then
  //rule 5
  for(superproperty in superproperties)
   if (not valuesToMatch.contain(superproperty) then
     output(null,triple(key.subject,rdfs:subPropertyOf,superproperty))
 else
  //rule 7
  for(superproperty in superproperties)
   if (!valuesToMatch.contain(superproperty) then
     output.collect(null,triple(key.subject,superproperty,key.object))
     newTriple.predicate = superproperty
     output(null,newTriple)
```

Algorithm 8 RDFS reasoning: third version, second job

```
map(key, value):
  //key: irrelevant
  //value: triple
  if (domain_schema.contains(value.predicate)) then
    newValue.flag = 0
    newValue.resource = value.predicate
    output.collect(value.subject, newValue)
  if (range_schema.contains(value.predicate)) then
    newValue.flag = 1
    newValue.resource = value.predicate
    output.collect(value.object, newValue)

reduce(key, iterator values):
    for (value in values) do
    if value.flag = 0 then
      //Check in domain
      types.add(domain_schema.get(value.resource))
    else
      types.add(range_schema.get(value.resource))

  for(type in types) do
    output.collect(null,
                   triple(key.resource, rdf:type, type))
```

Algorithm 9 RDFS reasoning: third version, third job

```
map(key, value):
  //key: irrelevant
  //value: triple
  output(value, key.isDerived)

reduce(key, iterator values):
   for (value in values)
    if (value = false) then
     exit

   output.collect(null, key)
```

5.4.4 Third job: cleaning up duplicates

The two jobs before can generate some duplicates against the input. With this job we clean up the derivation saving only unique derived triples. The structure of this job is simple. The mapper reads the triples and set them as the key of the intermediate tuples. As value it set true if the triple was derived or false in case it was original. For every triple we know whether it is derived or not because when we store them on disk we flag them in case they are from the input or derived.

The reducer simply iterates over the values and it returns the triple only if it does not find any "false" between the values, because that would mean that the triple was also in input. The algorithm is reported in Algorithm 9.

5.4.5 Fourth job: execute subclasses rules

The last job executes rules 8, 9, 11, 12 and 13. In this job the mapper is slightly different than the jobs before. In the previous cases the mappers check whether the input triples match with the schema before forwarding them to the reducers. This operation limitates the number of triples that are sent to the reducer but leaves the door open for duplication against the input.

Here we do not check the input triple but instead we forward everything to the reducer. In doing so we can also eliminate the duplicates against the input and consequently avoid to launch a second job that is cleaning up the duplicates. The disadvantage of this choice is that the reducer has to process all the data in input, even if the triple does not match with the schema. However this job works only with a subset of the data ("type" and "subclass" triples) and during our tests we noticed that it is faster if we let the reducer work a bit more than lightning this job and execute another job to filter the output.

For the rest the structure is similar than in the jobs before. We group the triples and then we proceeds to the derivation during the reduce phase. The re-

ducers load in memory the subclass triples and the sub-property of *rdfs:member*. The last triples are needed for the derivation of rule 12.

The rules 8, 12 and 13 do not require a join between two triples. What we do is simply checking if in the input or in the derived hashset we find a triple that correspond in one of the antecedent of the rules. If we find one, we check first whether the output we would derive is already present in the input (rule 12 using the *rdfs:member* sub-property triples) or if we would derive the same through rule 11 (we do this for rule 8 and 13, checking the subclass schema). In case all the conditions hold we can derive the new triple being sure that it will not be a duplicate with the input.

The algorithm is reported in Algorithm 10.

5.4.6 Fifth job: further process output special rules

As described in section 5.3.1 the rules 8, 12 and 13 can generate some output can could lead to another derivation. More in particular, with rule 8 we could derive that some classes are subclasses or *rdfs:Resource*. With rule 12 we can infer that some properties are sub-properties of *rdfs:member* and some other triples could inherit this relation on them. Also with rule 13 we can derive that some classes are subclasses of *rdfs:Literal*. These special cases are not common in the data that we crawled in the web and it is debatable if such triples are useful (like being a subclass of Literal) or not. We have implemented a job that derives this last bit of information, but since these rules are quite uncommon, the job starts only if the previous does derive some of them. In case the user is not interesting in this "special" derivation and wants to save time, this job can be skipped.

The structure of the job is different than the previous. To explain it let's take as example the following case. We could have these triples in input:

```
A rdf:type rdfs:ContainerMembershipProperty .
B rdfs:subPropertyOf A .
```

If we apply only rule 12 we will derive only that A is a sub property of *rdfs:member*. However, after we have derived it, rule 5 can also fire stating that also B is a sub property of *rdfs:member*.

In order to derive also this second statement the mappers first check whether the sub-property triple in input has *rdfs:member* as object or if it has as object a property that is sub-property of *rdfs:member*. In both cases they will forward the triple to the reducers setting as key the subject and as value the object. The reducers will check the values searching if they find *rdfs:member* between them. In case they find it, they will not output the statement because in the input it is already existing a statement like *key rdfs:subPropertyOf rdfs:member* and therefore we should not write it again. If it is not present we should proceeds writing the conclusion since we are in the case in which a sub-property triple's object matches a *rdfs:member* triple subject and the transitivity rule (rule 5) should be applied. The other rules works in a similar way. This job does

Algorithm 10 RDFS reasoning: third version, fourth job

```
map(key, value):
 //key: irrelevant
 //value: triple
 if (value.predicate = rdf:type then
   key.flag = 0
   key.value = value.subject
   output(key, value.object)

 if (value.predicate = rdfs:subClassOf then
   key.flag = 1
   key.value = value.subject
   output(key, value.object)

reduce(key, iterator values):
   for(value in values) do
    inValues.add(value)
   for(value in inValues) do
     superclasses.add(subclass_schema.recursive_get(value))

   for(superclass in superclasses) do
     if (!values.contains(superclass) then
       if (key.flag = 0) then //rule 9
        output.collect(
          null, triple(key.subject, rdf:type, superclass))
       else //rule 11
         output.collect(
          null, triple(key.subject, rdfs:subClassOf, superclass))

   //Check special rules
  if ((superclass.contains(rdfs:Class)
       or inValues.contains(rdfs:Class))
       and not subclass_schema.get(key.subject).contains(rdfs:Resource))
         then //rule 8
           output.collect(
             null, triple(key.subject, rdfs:subClassOf, rdfs:Resource)
   if ((superclass.contains(rdfs:ContainerMembershipProperty)
       or inValues.contains(rdfs:ContainerMembershipProperty))
       and not subprop_member_schema.contains(key.subject))
         then //rule 12
           output.collect(
             null, triple(key.subject, rdfs:subPropertyOf, rdfs:member)
  if ((superclass.contains(rdfs:Datatype)
       or inValues.contains(rdfs:Dataype))
       and not subclass_schema.get(key.subject).contains(rdfs:Literal))
         then //rule 13
           output.collect(
             null, triple(key.subject, rdfs:subClassOf, rdfs:Literal)
```

not generate any duplicate, because we forward all the relevant input triples, therefore there is no need to run another cleaning up job.

The algorithm is reported in Algorithm 11.

Algorithm 11 RDFS reasoning: third version, fifth job

```
map(key, value):
 //key: irrelevant
 //value: triple

 if value.predicate = rdfs:subPropertyOf then
   if value.object = rdfs:member or
      memberProperties.contains(value.object) then
    output.collect('1' + value.subject, value.object)

 if value.predicate = rdfs:subClassOf then
   if value.object = rdfs:Literal or
      literalSubclasses.contains(value.object) then
     output.collect('2' + value.subject, value.object)
   else if value.object = rdfs:Resource or
      resourceSubclasses.contains(value.object) then
     output.collect('3' + value.subject, value.object)

 if memberProperties.contains(value.predicate) or
      value.predicate = rdfs:member then
     output.collect('4' + value.subject, value.object)

reduce(key, iterator values):
  for (value in values)
   if value = rdfs:member or
      value = rdfs:Literal or
      value = rdfs:Resource then
    exit

  switch (key[0])
   case '1':
     output.collect(null,
       triple(key.resource, rdfs:subPropertyOf, rdfs:member)
   case '2':
     output.collect(null,
       triple(key.resource, rdfs:subClassOf, rdfs:Literal)
   case '3':
     output.collect(null,
       triple(key.resource, rdfs:subClassOf, rdfs:Resource)
   case '3':
     output.collect(null,
       triple(key.resource, rdfs:member, key.resource2)
```

Chapter 6

OWL reasoning as MapReduce functions

In this chapter we present an initial algorithm for OWL reasoning. We implement OWL reasoning using the rules proposed by the ter Horst fragment. These rules are more complicated to implement than RDFS ones because:

- some rules require a join between three or even four different triples. See for example rule 14b or 16;
- some rules require a join that is between two sets that both have many triples. That means we cannot load one side in memory as we did for the RDFS schema triples;
- the rules depend on each others and there is no rule ordering without a loop.

Because of this we cannot apply the optimizations we introduced for the RDFS reasoning and we must come up with other optimizations in order to present an efficient algorithm. The algorithm presented is at a primitive state, with only few optimizations introduced.

This chapter is organized as follows: first, in section 6.1, we will provide a complete overview of the algorithm, while later, in the following sections, we will explain more in detail every single part of it. This algorithm was implemented and tested on two small datasets. The performances are reported in section 7.2.3.

6.1 Overview

The rules proposed by the ter Horst fragment are reported in Table 2.2.

The rules are grouped in eight *blocks*, depending if they use the same schema triples, or in case they return a similar output.

Some of the jobs contained in the eight blocks are executed more than one time, because one single pass is not enough to compute a complete closure. Other job are executed only one time, but they produce duplicates and we need to launch an additional job to filter them out.

The rules execution consists in a main loop where the blocks are executed at least two times. We repeat the loop untill all the jobs do not derive any new triple. At that point we can be sure that we have computed a complete closure and we can stop the execution. Figure 6.1 reports a graphical depiction of the algorithm. From the figure we can see how the loop of eight blocks is executed until we stop deriving new statements.

In the next sections we are going to describe more in details each of these blocks. We will report the algorithms in pseudo code, excepts of the algorithm who deletes the duplicates because this algorithm is the same used during the RDFS reasoning and it was already presented in section 5.4.4.

6.2 First block: properties inheritance

The first rules we execute are rules 1, 2, 3, 8a and 8b. These rules concern the derivation of certain conclusions based on the properties of the predicates. For example if a predicate is defined as symmetric we can rewrite the triples with that predicate exchanging the subject and the object. Rules 1, 2 and 3 are not recursive. It means that their output cannot be reused as the rules input. Rules 8a and 8b are recursive but we can store the schema triples (p *inverseOf* q) in memory and recursively derive all the statements with one pass.

The algorithm works as follow. First the mappers load in memory the schema triples that are needed for the rules execution. These triples define the properties inverses and the list of the functional, inverse functional and symmetric properties. The mappers check if the input triples match with the schema and in case they do they are forwarded to the reducers.

The map algorithm outputs some tuples that encode the triples so that they limitate the number of derived duplicates. The algorithm attaches a flag to the key depending on which rule can be applied. This operation is done in a similar way than for the RDFS reasoning and aims to avoid the generation of duplicates.

The reducer algorithm starts first to load the values in a memory data structure, in order to remove eventual duplicates. After, it continues deriving the new statements according to the specific rule logic, using the information in the key and in the schema triples.

During this job we also process the triples that can fire rule 4. The purpose is to split the triples that can potentially fire rule 4 from the others. To do so the map algorithm loads in memory the schema triples for the transitive properties and it saves the triples that match against this schema in a special folder. Rule 4 will be executed in the next block but it requires we launch the job more times, therefore, if we filter the input triples at this stage, the next block will be much faster because it does not have to read all the input but only the subset

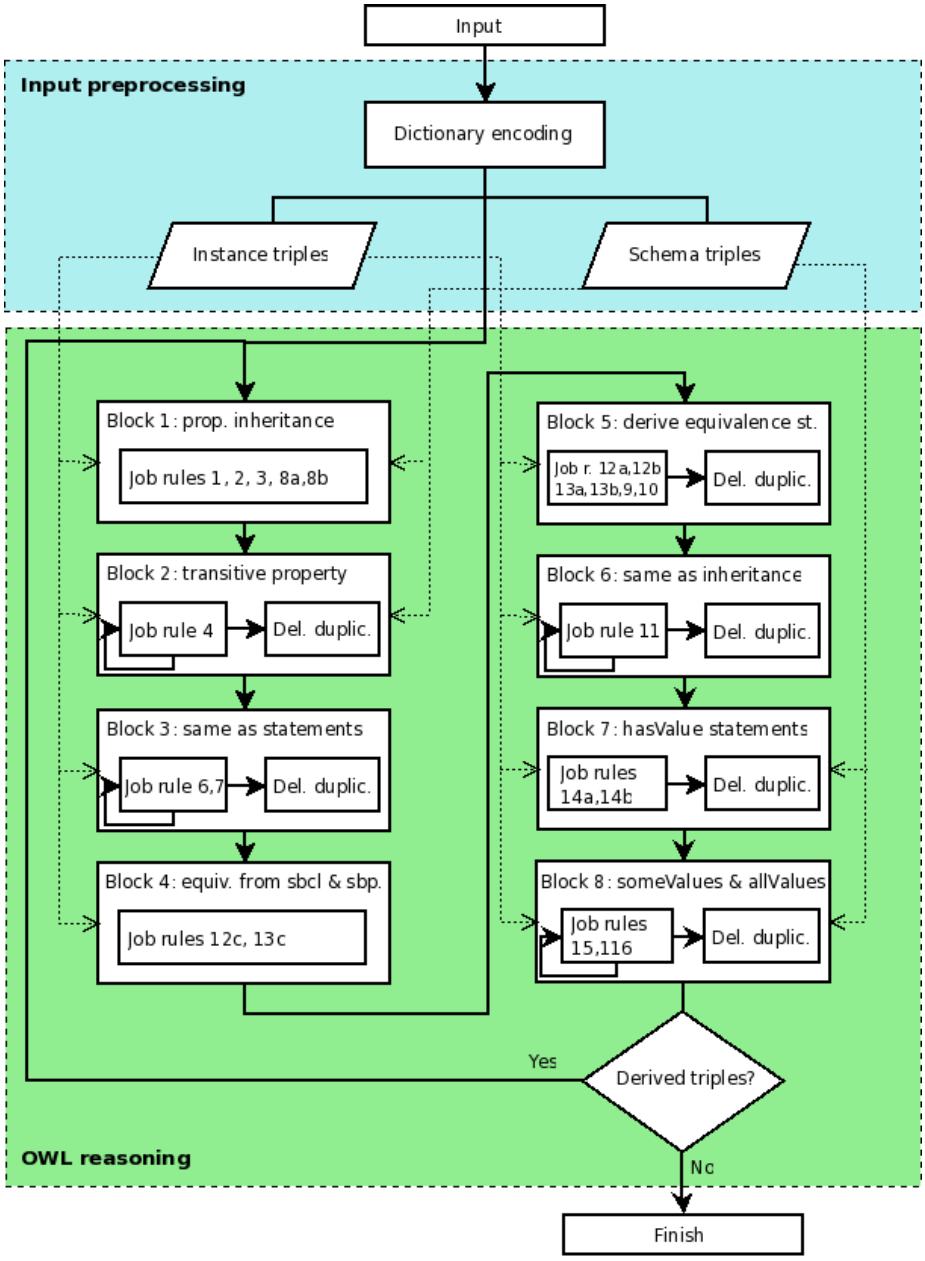

Figure 6.1: Overview of the OWL reasoner

Algorithm 12 OWL: first block, encode rules 1, 2, 3, 8a and 8b

```
map(key, value):
//key: irrelevant
//value: input triple
if (functional_prop.contains(value.predicate) then
  key = '0' + value.subject + value.predicate
  output.collect(key, value.object)

if (inverse_functional_prop.contains(value.predicate) then
  key = '0' + value.object + value.predicate
  output.collect(key, value.subject)

if (symmetric_properties.contains(value.predicate) then
  key = '1' + value.subject + value.object
  output.collect(key, value.predicate)

if (inverse_of_properties.contains(value.predicate) then
  key = '2' + value.subject + value.object
  output.collect(key, value.predicate)

//This doesn't encode the rule 4. It is used only to store the
//triples in another location for the following block
if (transitive_properties.contains(value.predicate) then
  key = '3' + value.subject + value.object
  output.collect(key, value.predicate)

reduce(key, iterator values):
  switch (key[0])
    case 0: //Rule 1 and 2
      values = values.unique
      for(value in values)
       for(value2 in values)
         output.collect(null, triple(value, owl:sameAs, values)
    case 1: //Rule 3
      for(value in values)
         output.collect(null, triple(key.object, value, key.subject)
    case 2: //Rule 8a,8b
      for(value in values)
         resource1 = key.subject
         resource2 = key.object
         p = value
         begin:
         if (p_inv = inverse_of.get(p) and output.not_collected(p)) then
             output.collect(null, triple(resource2, p_inv, resource1)
             swap(resource1, resource2)
             p= p_inv
             goto begin
    case 3:
      //Just forwards the triples for the next block
      //(rule 4) in a special directory
      for(value in values)
         output.collect(null, triple(key.subject, value, key.object))
```

we have prepared this time.

After we have executed this job we do not execute a job to clean up the duplicates because even if it can produce some duplicates they are supposed to be rare and they do not justify the execution of a job that needs to parse all the input to clean few duplicates. A cleaning job will be executed at the end of the next block considering also the output of this block. The algorithm is presented in Algorithm 12.

6.3 Second block: transitive properties

The second blocks executes only rule 4. The join is done in a "naive" way because we are unable to load one of the two parts in memory and do the join on the fly.

The rule is encoded so that the join is done only between triples that share one term. Since there can be chains of triples with transitive properties, we need to launch this job more times, till we have derived everything out of it. We do not use as input the all data set but instead only the content of the special folder that was prepared during the previous job. This folder contains only the triples that match with the schema, that are a small subset of the data.

The mapper outputs as key the URIs that can be used as a match point, that are either the subject or object of the input triples. As value the mapper sets the other resource, adding also a flag that indicates the position of the two resources within the triple.

The reducer iterates over the values and loads them in two different sets, depending on their position within the triple. After, if both sets have some elements, the reducer proceeds returning all the combinations between the two in-memory sets.

This operation is repeated until the amount of derived triples does not increase anymore. At this point we launch a cleaning job that filters out all the duplicates that were generated during the previous executions. The algorithm is reported in Algorithm 13.

6.4 Third block: sameAs statements

The third job executes rule 6 and 7. This job is similar to the previous one and it also needs to be repeated more times since there can be chains of "sameAs" triples that require the job being repeated. This job generates a considerable amount of duplicates, because everytime it is launched it will derive at least the same amount of information that was derived before. For this reason we need to launch a cleaning job right after it.

The two rules are both executed in a "naive" way. The map returns both the subject and the object as the tuple's key. The reducer simply loads in memory all the values and returns all the possible sameAs statements between

Algorithm 13 OWL: second block, encode rule 4

```
map(key, value):
//key: irrelevant
//value: input triple

if (transitive_properties.contains(value.predicate) and
    value.subject != value.object then
   key = value.predicate + value.object
   output.collect(key, '0' + value.subject)
   key = value.predicate + value.subject
   output.collect(key,'1' + value.object)

reduce(key, iterator values):
   values = values.unique
   for(value in values)
      if (value[0] == 0) then
        join_left.add(value.resource)
      else //value[0] = 1
        join_right.add(value.resource)

  for(left_element in join_left)
    for(right_element in join_right)
      output.collect(null,
       triple(left_element,key,key.predicate,right_element)
```

Algorithm 14 OWL: third block, encode rules 6 and 7

```
map(key, value):
//key: irrelevant
//value: input triple
  output.collect(value.subject, value.object)
  ouput.collect(value.object, value.subject)

reduce(key, iterator values):
  values = values.unique
  for(value in values)
   output.collect(null, triple(key, owl:sameAs, value)
   for(value2 in values)
     output.collect(null, triple(value, owl:sameAs, value2)
```

the resource in the key and the values (this is the output of rule 6) and between the elements in the values (this is the output of rule 7).

There are problems of scalability since we need to store all the triples in memory. In case there is one very common resource we may not be able to store them in memory, and therefore we are unable to proceed with the reasoning task. So far we could not come up with an optimization to avoid this problem and indeed these two rules revelead to be problematic during our tests so that we needed to deactivate them.

The algorithm is reported in Algorithm 14.

6.5 Fourth block: equivalence from subclass and subproperty statements

The fourth job executes rules 12c and 13c. These two rules are not recursive, and this means we can execute this job only one time.

The job is relatively fast compared to the others, because it accepts in input only schema triples that are either "subclass" or "subproperty" relations.

The two rules fire when there are two triples that have the same subject and object in the opposite positions. If we want to group them together we must encode the subject and the object in the key, but we need that two triples with subjects in different positions objects are grouped together. What we do is to write the subject and the object in the tuple's key depending on their number. If the subject has a number that is greater than the object it will be the key as the sequence *subject+object*. In the opposite case the key will become *object+subject*. In order to know what is the position of the first resource in the key we set as value either 1 in the first case or 0 in the second case.

With this methodology in case there are two triples that have the object equals to the other's subject and viceversa, they will be encoded in two tuples

Algorithm 15 OWL: fourth block, encode rules 12c and 13c

```
map(key, value):
//key: irrelevant
//value: input triple
 if (value.subject < value.object) then
   if (value.predicate == rdfs:subClassOf) then
     key = 'SC' + value.subject + value.object
   if (value.predicate == rdfs:subPropertyOf) then
     key = 'SP' + value.subject + value.object
   output.collect(key,0)
 else
   if (value.predicate == rdfs:subClassOf) then
     key = 'SC' + value.object + value.subject
   if (value.predicate == rdfs:subPropertyOf) then
     key = 'SP' + value.object + value.subject
  output.collect(key,1)

reduce(key, iterator values):
  values = values.unique
  if (values.contains(0) and values.contains(1)) then
    if (key[0] = SC) //subclass
      output.collect(
       null,triple(key.resource1,owl:equivalentClass,key.resource2)
      output.collect(
       null,triple(key.resource2,owl:equivalentClass,key.resource1)
    else //subproperty
      output.collect(
       null,triple(key.resource1,owl:equivalentProperty,key.resource2)
      output.collect(
       null,triple(key.resource2,owl:equivalentProperty,key.resource1)
```

with the same key but with two different values.

The reducer algorithm simply checks whether the tuples have different values. In case they have it outputs a new triple. The algorithm is reported in Algorithm 15.

This job can generate some duplicates against the data in input, therefore we must launch another job to clean up the duplicates.

6.6 Fifth block: derive from equivalence statements

This job executes rules 9, 10, 12a, 12b, 13a and 13b. All these rules output triples that are either about subclass or about subproperty relations.

Rules 12a, 12b, 13a and 13b have only one antecedent and their implementation is straightforward. The map algorithm outputs the subject as key and the object as value. The reduce algorithm filters out the duplicates and returns the corresponding derived triple.

Rules 9 and 10 require a join between the "type" triples and the "sameAs" triples. The map algorithm outputs the intermediate tuples setting as key the resource that could be matched. The reduce algorithm iterates over the values and stores the "sameAs" triples in a in-memory structure. As soon as it encounters one triple that has as type either "owl:Class" or "owl:Property" it derives a new triple for each of the elements in the in-memory data structure. The algorithm sets a flag so that the next "sameAs" values are not stored but instead used immediately for the new derivation and then discarded.

All the rules encoded are not recursive and therefore we do not need to repeat the job. However the job produces some duplicates and therefore we must run a cleaning job right after. The algorithm is reported in Algorithm 16.

6.7 Sixth block: same as inheritance

The sixth block encodes rule 11. This rule is problematic because it requires a join between two datasets that have many elements. If we encode the join in the naive way we will need to store both sets in memory but it would likely not fit. Consider the example:

```
url1 rdf:type Link
url2 rdf:type Link
...
urln rdf:type Link
Link owl:sameAs URL
```

The "naive" way requires that we load all the values in memory because the reducer has access to the values only through an iterator and it cannot scroll it more than one time. Therefore we are obliged to store all the "type" values in

Algorithm 16 OWL: fifth block, encode rules 9, 10, 12a, 12b, 13a, 13b

```
map(key, value):
//key: irrelevant
//value: input triple
 if (value.predicate = owl:equivalentClass) then
  output.collect(value.subject, '0' + value.object)
 if (value.predicate = owl:equivalentProperty) then
  output.collect(value.subject, '1' + value.object)
 if (value.predicate = rdf:type and value.object = owl:Class) then
  output.collect(value.subject, '2' + value.object)
 if (value.predicate = rdf:type and value.object = owl:Property) then
  output.collect(value.subject, '3' + value.object)
 if (value.predicate = owl:sameAs) then
  output.collect(value.subject, '4' + value.object)

reduce(key, iterator values):
 values = values.unique
 for(value in values)
   if (value[0] = 0) //Rule 12a, 12b
     output.collect(null, triple(key,
                                 rdfs:subClassOf, value.resource)
     output.collect(null, triple(value.resource,
                                 rdfs:subClassOf, key)
   if (value[0] = 1) //Rule 13a, 13b
     output.collect(null, triple(key,
                                 rdfs:subPropertyOf, value.resource)
     output.collect(null, triple(value.resource,
                                 rdfs:subPropertyOf, key)
   if (value[0] = 2)
     subClassFlag = true
     //empty elements cached
     for(sameResource in sameAsSet)
        output.collect(null, triple(key,
                                    rdfs:subClassOf, sameResource)
   if (value[0] = 3)
     subPropertyFlag = true
     //empty elements cached
     for(sameResource in sameAsSet)
        output.collect(null, triple(key,
                                    rdfs:subPropertyOf, sameResource)

   if (value[0] = 4)
     if (not subClassFlag || not subPropertyFlag)
        sameAsSet.add(value.resource)
     else if (subClassFlag) //Rule 9
               output.collect(null, triple(key, rdfs:subClassOf
                                           value.resource)
          else //Rule 10
               output.collect(null, triple(key, rdfs:subPropertyOf,
                                           value.resource)
```

memory until we encounter one or more "sameAs" values. Obviously if there are too many elements in the group (consider the example above) the algorithm is impossible to execute.

To solve this problem we should put all the "sameAs" tuples at the beginning of the iterator. The MapReduce programming model does not provide such feature, but we can exploit the characteristics of the Hadoop framework to sort the tuples in such a way that all the "sameAs" tuples appear as first in the list.

In the Hadoop framework, the intermediate tuples returned by the mappers are first stored locally and then sent to the machines that execute the reduce tasks. All the tuples with the same key should be sent to the same node because they must be grouped together and processed by the reduce algorithm. Hadoop uses the hash code of the tuple's key to identify which node it should send the tuple to. For example if there are 10 reducers tasks to execute, the mappers calculates the hashcode of the tuple's key, it divides it by 10 and according to the module it sends the tuple to the correct node. In this way all the tuples with the same key will end in the same node.

After one node has received all the tuples from the mappers it sorts them in a list so that all the tuples with the same key are next to each others. Then it calls the user-defined reduce function passing as argument the first key of the list and an iterator over the values of the list of tuples. Every time that the user calls the "iterator.next" function, the framework checks the next tuple in the list, and in case it differs, it stops because that means that the group is finished and a new group starts on the next record.

The framework gives the possibility to the user to overwrite three standard functions: the first is the function that calculates the hash code, the second is the function that sorts the tuples in the list and the last is the function that checks whether the next key is the same than the previous when we call the function "iterator.next".

We rewrite these functions so that:

- during the map phase, we set as key the URI that should be matched in the join plus a flag, 'S' if the triple is sameAs or 'W' if it is a generic one;
- we design the hash-code function as a function that calculates the hash-code on the key *without* the last flag
- we design the function that sorts the tuple's key as a function that sorts the tuples considering the keys *with* the last flag. This function will always put the "sameAs" triples before than others because 'S' < 'W'
- we design the function that checks the next value on the list as a function that compares the keys *without* considering the last flag.

Let's consider an example to see what happens. We have these triples in input:

```
a rdf:type Class
Class owl:sameAs AnotherClass
```

The mapper will output the tuples:

```
...
<ClassW, 2rdf:typeClass>
<ClassS, 0AnotherClass>
...
```

The two tuples have different keys, therefore they can be partitioned in two groups that will be processed by different nodes. However, we have set a function that calculates the hash-code without considering the last character and therefore they will be both partitioned into the same group.

After the reducer has sorted the tuples, the "sameAs" triples will always be the first in the list because they share everything except the last character that is either 'S' or 'W' ('S' < 'W'). When in our reduce function we call the function "iterator.next" the comparison will not consider the final byte of the key and it will return the next tuple as equal, even if the two have different keys.

The purpose of this trick is to have the "sameAs" tuples at the beginning of the list so that we can load them in memory. The reduce algorithm starts scrolling the iterator and it stores only the "sameAs" values in memory. After, it proceeds executing the join between all the values it has stored in memory and each of the tuples it will fetch from the iterator. The flags on the tuple's values are needed because we need to know what kind of tuple we are processing, "sameAs" or generic, and in case it is generic, whether the key was the subject or the object.

The algorithm is reported in Algorithm 17. We need to execute this job two times. The job generates some duplicates and therefore we need launch a third filtering job.

6.8 Seventh block: hasValue statements

In this block we execute rules 14a and 14b. Both rules have access to the schema triples that define the relations "onProperty" and "hasValue". Rule 14a works only with "type" triples, while the other rule works with generic ones. We assume that rule 14b cannot accept "type" triples because this would mean that we define a relation "onProperty" on the *rdf:type* predicate and since we do not allow any redefinition of the standard constructs, this case is simply ignored. This implies the two rules works on two disjoint groups.

The mappers split the triples in two groups depending whether the predicate is *rdf:type* or not. If the triple has *rdf:type* as predicate it will be processed according to the logic of rule 14b, otherwise it will be by rule 14a. We set as the tuple's key the subject of the triple so that the tuples are grouped according to the subject. This grouping aims to limitate the number of generated duplicates. In case of rule 14b we put as value only the object, since the predicate is always *rdf:type*, while in the other case we encode as value both the predicate and object.

Algorithm 17 OWL: sixth block, encode rule 11

```
map(key, value):
//key: irrelevant
//value: input triple
 if (value.predicate = owl:sameAs) then
   key =  value.subject + 'S'
   output.collect(key,'0' + value.object)
   key = value.object + 'S'
   output.collect(key,'1' + value.subject)
 else
   key = value.subject + 'O'
   outValue = '1' + value.predicate + value.object
   output.collect(key,outValue)
   key = value.object + 'O'
   outValue = '2' + value.predicate + value.subject
   output.collect(key,outValue)

reduce(key, iterator values):
  value = values.unique
  for(value in values)
    if (value[0] = '0') then //sameAs triple
      sameAsSet.add(value.resource)
    else
      for(sameResource in sameAsSet)
        if (value[0] = '1') then //value is p + o
          output.collect(null,triple(sameResource,
                                     value.resource1,value.resource2)
        else //value is p + s
          output.collect(null,triple(value.resource2,
                                     value.resource1,sameResource))
```

Algorithm 18 OWL: seventh block, encode rules 14a,14b

```
map(key, value):
  //key: irrelevant
  //value: input triple
  if (value.predicate = rdf:type) then
    output.collect(value.subject,'0'+value.object)
  else
    output.collect(value.subject,'1'+value.predicate+value.object)

reduce(key, iterator values):
  for(value in values)
    if (value[0] = 0) then //rules 14b
      onProperties = onPropertySchema.get(value.object)
      hasValues = hasValueSchema.get(value.object)
      if (onProperties != null and hasValues != null) then
        //the triple's object matches with the schema
        for(onProperty in onProperties)
            for(hasValue in hasValues)
                output.collect(null,triple(key.subject,onProperty,hasValue)
    else //rule 14a
      onProperties = onPropertyInvertedSchema.get(value.predicate)
      hasValues = hasValueInvertedSchema.get(value.object)
     types = onProperties $\and$ hasValues
     for(type in types)
        output.collect(null,triple(key.subject,rdf:type,type)
```

The reducers first check the flag in the value to know which rule they should execute. After, they check if the information contained in the values matches against the schema. In case it does, the reducers derive the new triples.

The two rules use the same schema triples for the join, but using different match points. For this reason the two schema triples ("onProperty" and "hasValue") are loaded in 4 different hashtables. In the first two we set the triples subjects as key and the objects as value. These will be used by rule 14b. The others are generated in the opposite way, using the triples objects as key and the subjects as value. These two will be used for rule 14a.

The algorithm is reported in Algorithm 18. This job generates some duplicates and therefore we launch another job to clean up the duplicates.

6.9 Eighth block: someValues/allValuesFrom schema

In this block we execute the last two rules: rule 15 and rule 16. These two rules require a join between 4 different triples. Two of them are schema triples and they can be loaded in memory. The other two are data triples and therefore we

need to execute the join in a "naive" way.

As usual we use the map phase to group the triples to limitate the duplicates. Since the join involves also two parts with many elements (the "type" triples and the generic ones) we apply the same trick than we used with block 6 to avoid to load in memory all the elements. We add one byte to the tuple's key (the flag 'T' or 'W'), and then, using a special hash-code and sorting function, we are able to sort the list of values putting the "type" triples first.

In case the triple is a "type" one, it can be matched only by the object, therefore the mapper will output only one tuple with the subject as key and the object as value. In case the triple is a generic one, it can be matched using either the combination "subject + predicate" (rule 16) or the combination "object + predicate" (rule 15). Therefore, the map algorithm will output two tuples, one with the subject as key and the sequence "predicate+object" as value, and the other with the object as key and the sequence "predicate+subject" as value.

The reduce algorithm loads first the "type" triples in memory scrolling over the values. Then it checks the flag of the values and applies the appropriate rule according to it. In case the flag is 1 the algorithm will apply the logic of rule 16, otherwise it will apply the one for rule 15.

In case of rule 15 the reducer first checks whether the predicate that is contained in the value matches with the "onProperty" schema. When it does it further continues checking whether the "onProperty" values just derived match with the "someValuesFrom" schema. The values that are retrieved from this last matching are further checked against the "type" triples loaded before in memory. If there are values for which all these matching succeed, the algorithm derives the new triples.

In case of rule 16 the algorithm first matches the predicate against the "onProperty" schema. The values just returned from this join are further matched against the "allValues" schema and the output values against the "type" triples. Similarly as before, when all these matching succeed the algorithm derives the new statements.

Both rules are recursive and this means we must run the job more times until we have derived everything. After, we launch a job to clean up the duplicates that were produces during the previous jobs.

After this last block has finished the computation the main algorithm controls whether it has derived some new triples during the executions of the eight blocks. In case it did, it starts the loop again executing the first block.

Algorithm 19 OWL: eighth block, encode rules 15,16

```
map(key, value):
  //key: irrelevant
  //value: input triple
  if (value.predicate = rdf:type) then
    outKey = value.subject + 'T'
    outValue = '0' + value.object
    output.collect(outKey,outValue)
  else
    outKey = value.subject + 'W'
    outValue = '1' + value.predicate + value.object
    output.collect(outKey,outValue)

   outKey = value.object + 'W'
   outValue = '2' + value.predicate + value.subject
   output.collect(outKey,outValue)

reduce(key, iterator values):
  //''types'' will contain all the types of the resource in the
  //key
  types.clear();
  while (values hasNext and value[0] = 0)
     value = values next
     types.add(value)

  while(values hasNext)
     value = values next
     //match the predicate with the "onProperty'' schema
     onProperties = onPropertySchema.get(value.predicate)
     for(onProperty in onProperties)
       if (value[0] = 1) then //rule 16
         if (types.contains(onProperty) then
           allValues = allValuesSchema.get(onProperty)
           for(allValue in allValues)
              output.collect(null,
                        triple(value.subject,rdf:type,allValue)
       else //rule 15
           someValues = someValuesSchema.get(onProperty)
           for(someValue in someValues)
              if (types.contains(someValue) then
                output.collect(null,
                        triple(value.subject, rdf:type, onProperty)
```

Chapter 7

Results

In this chapter we report an analysis of the performances of the algorithms we have proposed. Section 7.1 reports some technical characteristics of the environment in which we conducted the tests. Section 7.2 reports the performances obtained and an evaluation of them.

7.1 Experiments settings

For the evaluation of the program we have used the cluster DAS3 at the Vrije Universiteit [1]. It is a cluster with 85 nodes, each equipped with two dual core processors with 4GB of main memory and 250GB hard disk. The nodes are interconnected through Gigabit Ethernet.

We set up an Hadoop cluster reserving one node for the jobtracker and one node for the namenode. We used a variable amount of slaves for the scalability tests, starting from 1 to 64.

The programs that were developed are written in Java 1.6 and they use the 0.19.1 version of Hadoop. The programs are publicly available on Internet [2].

7.2 Results

We have divided this section in three parts. Subsection 7.2.1 reports the performances and an evaluation of the dictionary encoding algorithm that was proposed in 4.4. Subsection 7.2.2 reports the performances and an evaluation of the RDFS reasoner. Subsection 7.2.3 reports the performances and a brief analysis of the OWL reasoner.

[1] http://www.cs.vu.nl/das3

[2] `https://code.launchpad.net/~jrbn/+junk/reasoning-hadoop`

7.2.1 Dictionary encoding performances

The dictionary encoding algorithm uses two MapReduce jobs to encode the data in a new more compact format. Our algorithm shows load balancing problems that are limited by the fact we use the triples IDs instead of the full text, but still persist with a big input. To solve this problem we introduced a cache with the most popular resources to help the load balancing. However, in case we decide to use the cache, we need to launch first an additional job to build the cache. The time that this job takes should be detracted by the advantage we gain in using it during the dictionary encoding.

What we require from the algorithm, whether we decide to use the cache or not, is to be scalable. We have measured the scalability of the algorithm with a different input size and with different nodes.

We report in Figure 7.1 the results of the scalability test varying the input size with and without the cache. Analogously, in Figure 7.2 we report the performances of our algorithm keeping the input size fixed but using a different number of nodes.

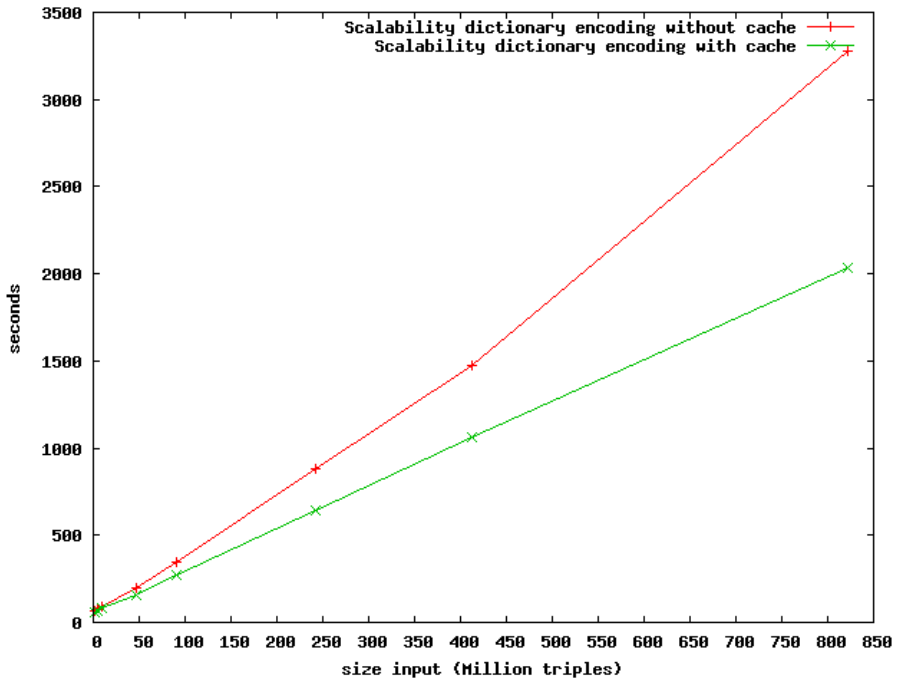

Figure 7.1: Scalability test of dictionary encoding varying the input size

From these graphs we can see how the algorithm performs better when we use the cache. This behavior is expected because without cache the computation speed is affected by the load balancing problem. When we introduce the cache, we eliminate or at least reduce the load balancing issue and the scalability shows

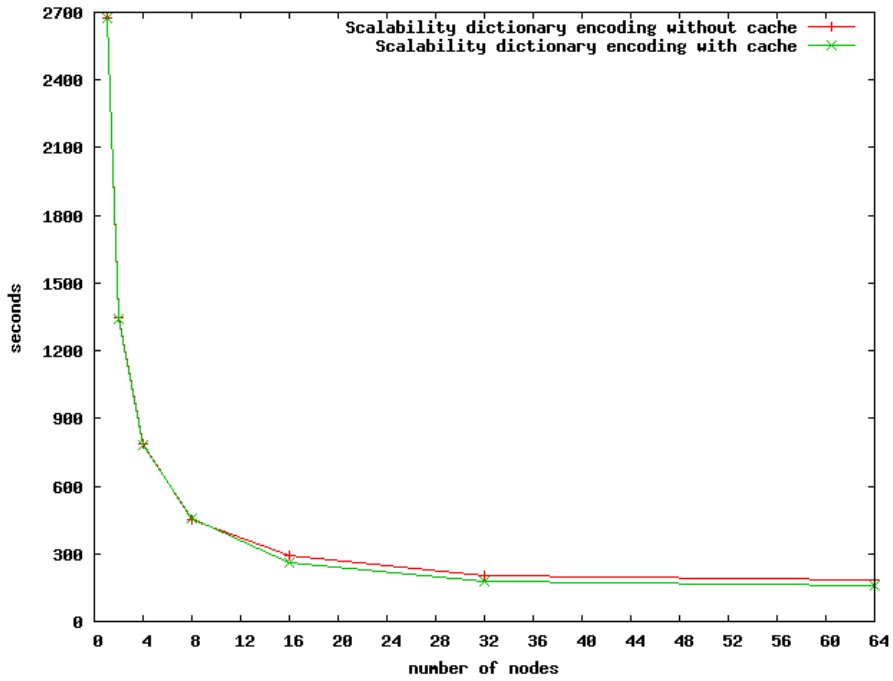

Figure 7.2: Scalability test of dictionary encoding varying the number of nodes

an almost perfect linearity.

In order to use the cache during the dictionary encoding we first must launch a simple MapReduce job that counts the most popular resources in the input. We require scalability also from this job, otherwise it cannot be used for large input. For this reason we have measured the performances of this job with different input sizes and the results are reported in figure 7.3. We conclude that also this job scales well because it shows a linear scalability.

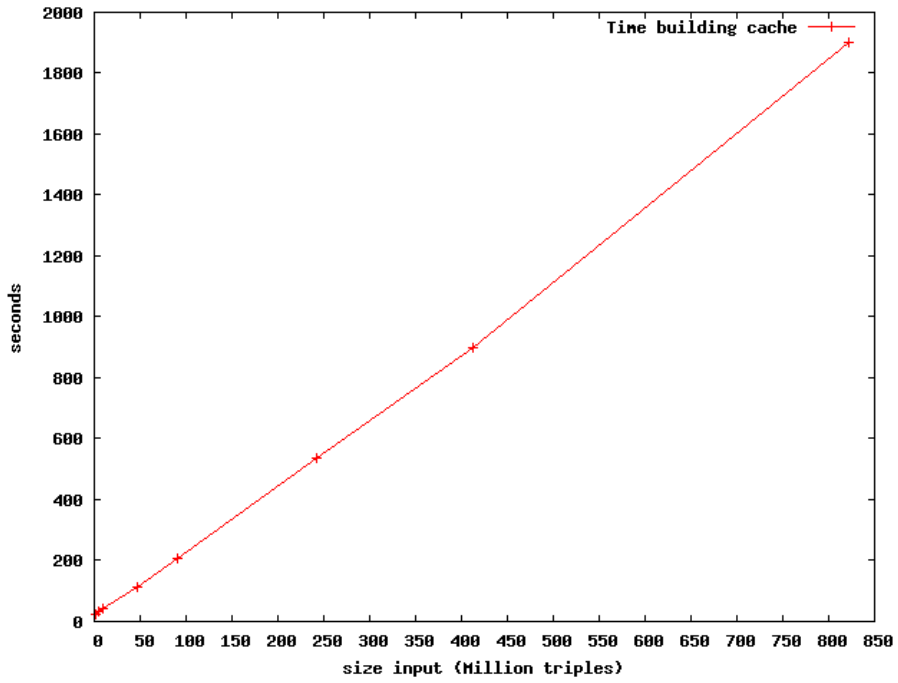

Figure 7.3: Scalability of building the cache

So far we have seen that the cache increases the performances of the algorithm both in terms of execution time and in terms of scalability. Also the algorithm that builds the cache proved to be scalable. In Figure 7.4 we summed up the time of building the cache with the time of executing the dictionary encoding and we compared it with the execution time of the algorithm without the cache. The purpose is to evaluate whether is worth or not to use the cache in terms of total execution time. What we can see is that the sum of the two executions is always longer than just executing the algorithm without using the cache. What we conclude, from these results, is that though the cache improves the scalability of the algorithm, it is not worth to use because the total execution is longer than just using the algorithm without the cache.

An explanation for it could be that in our case the load balancing does not hurt so much to justify a cache. However it can also be that the job of

building the cache is in general too slow and it will never be smaller than the gain obtained by using the cache.

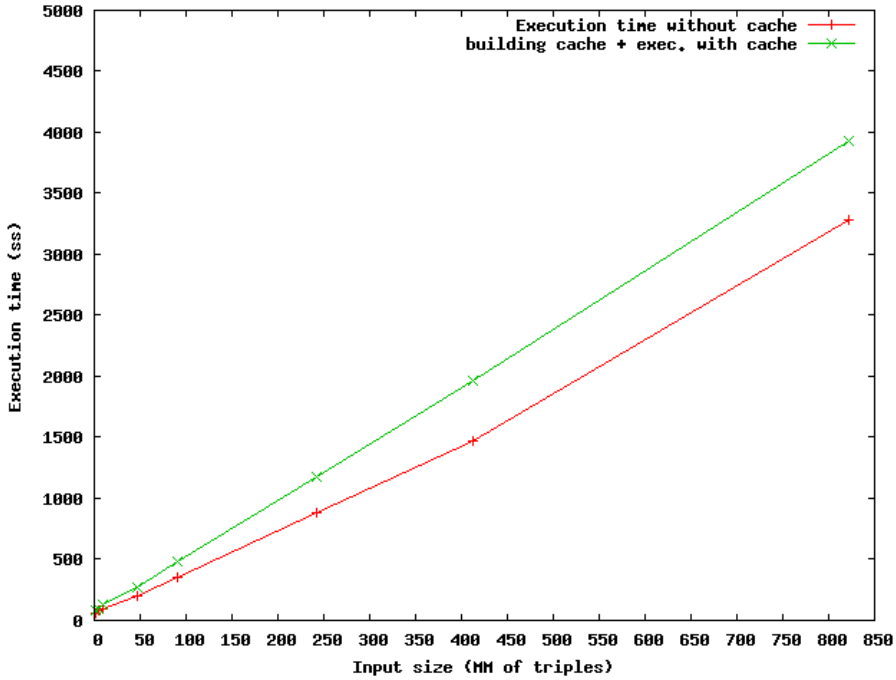

Figure 7.4: Performances between not using cache and using it considering also the cache's building time

One last analysis can be done varying the cache size and seeing if there is a notable increase of the performances if we use a bigger cache. For this purpose we launched the first job of dictionary encoding using a different cache's size. We expect that if we increase the cache size we obtain better performances until a certain limit. After, there shouldn't be any notable difference, because the job should be balanced enough. We conducted the tests and we reported the results in Figure 7.5. As it can be seen from the graph, the results reflect the initial expectations: the execution time decreases drastically until 15 elements, then the time decreases less and it stabilizes between 50 and 100 elements. We conclude from it that the ideal cache size for this input should be between 10 and 50 elements because after we do not gain any speedup anymore.

7.2.2 RDFS reasoning performances

We considered three aspects in evaluating the performances of the RDFS reasoner.

The first is correctness. Obviously the algorithm must produce a result that

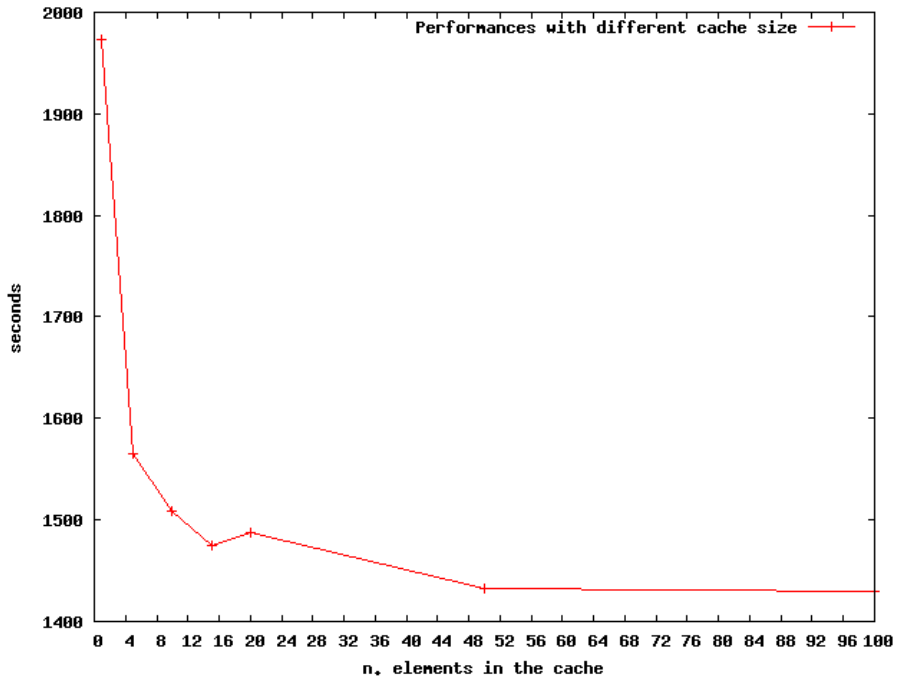

Figure 7.5: Performances using different cache's size

Dataset	Input	Output	Exec. time	Rel. StDev
Wordnet	1,942,887	3,004,496	2m25s	9.8%
DBPedia	150,530,169	21,447,573	2m44s	4.1%
Falcon	32,512,291	832,412,298	4m35s	1.1%
Swoogle	78,839,999	1,431,765,550	6m33s	0.7%
1B competition	864,846,753	29,170,100,223	58m43s	1.9%

Table 7.1: Performances RDFS reasoner on different datasets

is sound and complete and our first concern was to verify it. We first produced some test cases and verified that the output was correct. Then we compared the amount of derived triples of a small ontology (wordnet) with the amount that was derived by an already existing algorithm to see if there were some noticeable differences. We could not find any mistake, therefore we will assume that the result produced is correct, though we cannot guarantee that the implementation is bug-free.

The second aspect is stability. Since we use a distributed system, it is important that our algorithm is reliable, always ending at about the same time. For this reason we repeated every test three times and we picked the average and standard deviation of the outcome.

The third aspect is the scalability. The program we developed must be scalable, because this is our request. It is difficult to evaluate the scalability against the input size because, as it will be shown, much depends on the data itself. We tested the reasoner on different datasets of various sizes and we have reported the measurements. We have also tested the performances using the same dataset as input and changing the amount of nodes of the framework. First we started with 1 node, then we added another node, and so on, till we arrived to 64 nodes.

Here we present the results that we have collected.

Scalability with different data sets

As first we present the measurements of the reasoner using different datasets. We conducted the following tests using 32 working nodes.

Table 7.1 reports the performances obtained launching the algorithm on different data sets. From the table we notice that the deviation is relatively low, therefore we can conclude that our approach is reliable since it always terminated at about the same time.

In Figure 7.6 we report a chart were we indicate how much time every single job of the reasoner took for the execution. In table 7.2 we report how many triples every single job has processed in input and in output plus the ratio between them. We can see that in general Job 4 is the one that takes the most time for the computation and the one that generates most of the derivation.

The execution time does not depend on the input size. We plot in Figure 7.7 the execution time against the input. We can see from the plot that there

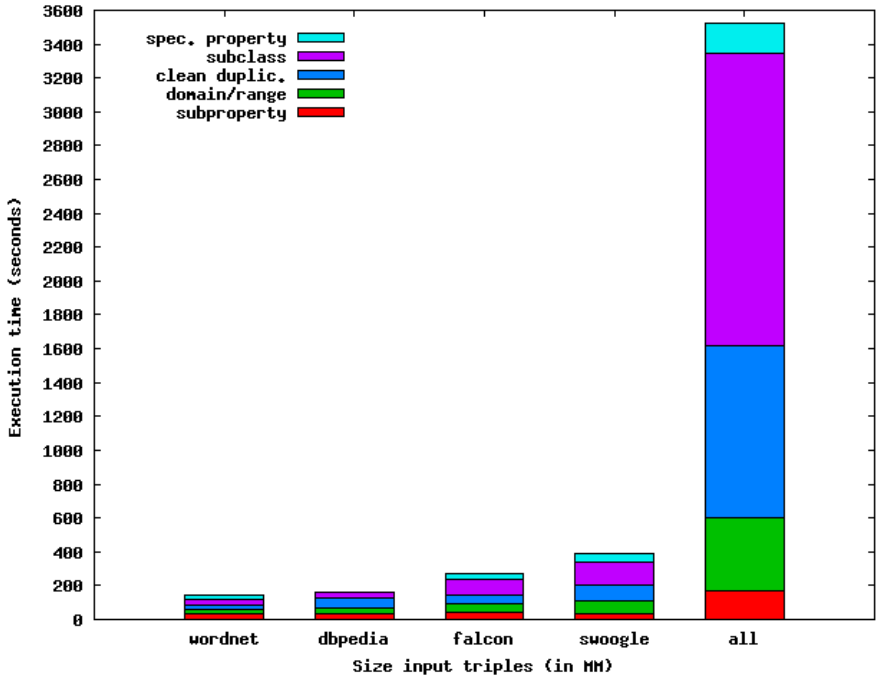

Figure 7.6: Scalability RDFS reasoner on different datasets

	Job 1			Job 2			Job 3			Job 4			Job 5		
Dataset	**I**	**O**	**R**	**I**	**O**	**R**	**I**	**O**	**R**	**I**	**O**	**R**	**I**	**O**	**R**
Wordnet	2M	0.29M	**0.15**	2.2M	0.896M	**0.4**	3.1M	1M	**0.34**	1.2M	1.9M	**1.55**	1.7M	0	**0**
DBPedia	150M	0	**0**	150M	3.6M	**0.02**	154M	2.3M	**0.01**	10M	19M	**1.87**	0	0	**0**
Falcon	32.5M	15M	**0.46**	47.6M	71M	**1.49**	119M	74.5M	**0.62**	71M	757M	**10.6**	39M	0.037M	**0**
Swoogle	78.8M	30M	**0.38**	109M	214M	**1.95**	323M	235M	**0.72**	219M	1.2B	**5.44**	104M	0.059M	**0**
1B Chal.	864M	387M	**0.44**	1.2B	1.6B	**1.33**	2.9B	1.9B	**0.64**	1.6B	27B	**16.54**	1B	0.1M	**0**

Table 7.2: RDFS reasoner

is any proportion between them.

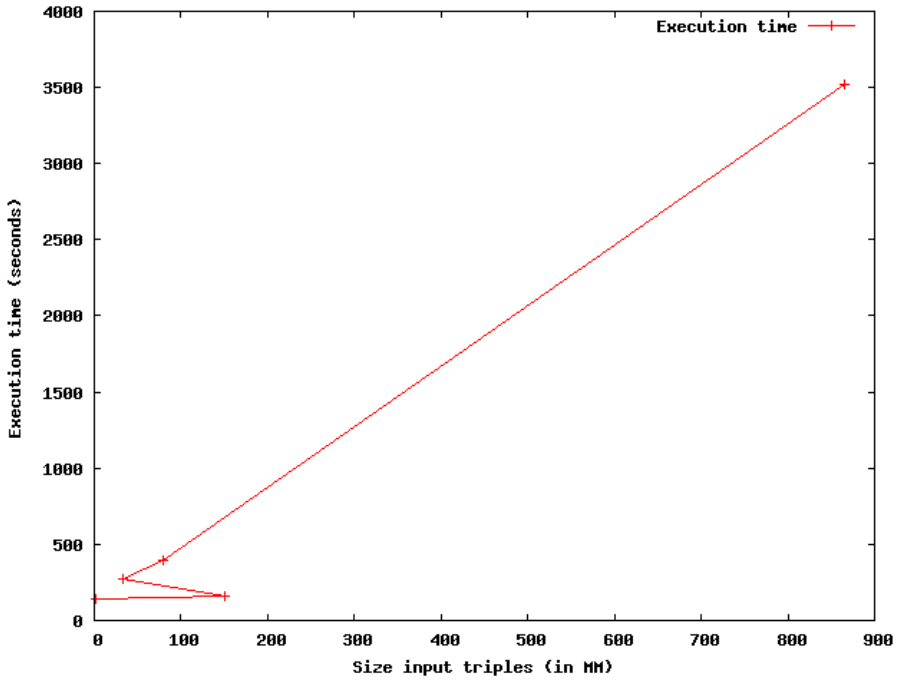

Figure 7.7: Scalability RDFS reasoner against input

Figure 7.8 reports the same graph but against the output triples. Here we see a linearity between the time and the amount of triples. This behavior is expected since the complexity of reasoning is not proportional to the input size, but instead is more related to the output size.

Scalability with different amount of nodes

We report now the scalability of the algorithm changing the number of computational nodes. We have used two fixed datasets, Falcon and DBPedia, and we started from 1 node and increased to 64. The results are reported in Figure 7.9 while table 7.3 reports the speedup and the efficiency for both datasets.

We notice from the graph that the execution time steadily decreases till 8 nodes it it stabilizes around 32 nodes. The explanation for this flattering is that with 32 or more nodes the computation load per node is not big enough to cover the framework overhead. In other words, the datasets are too small for so many nodes and we pay too much in terms of framework's overhead that there is no notable difference in the execution time.

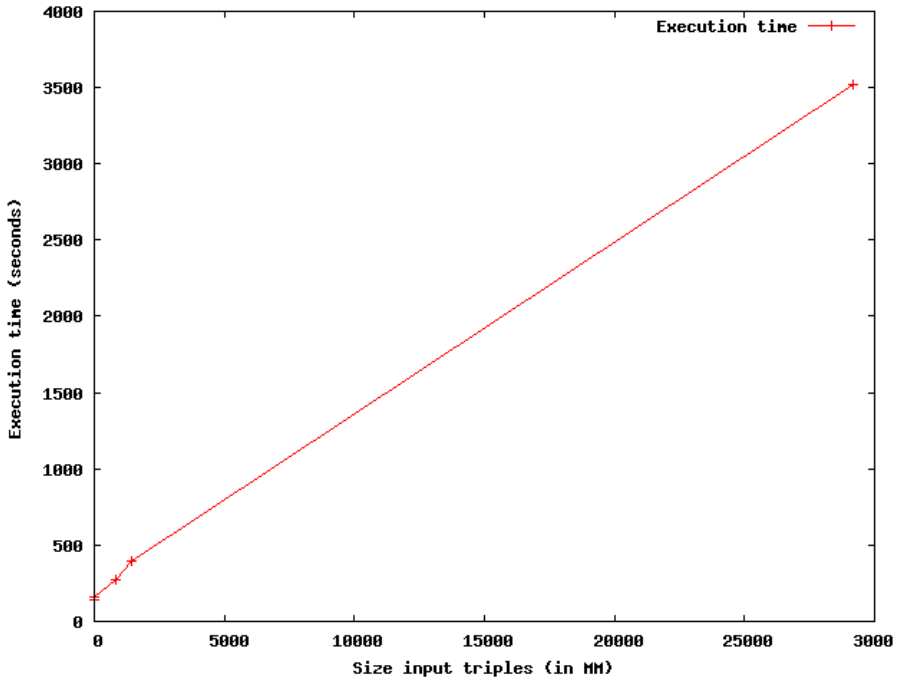

Figure 7.8: Scalability RDFS reasoner against output

Falcon				DBPedia			
Nodes	**Time**	**Speedup**	**Efficiency**	**Nodes**	**Time**	**Speedup**	**Efficiency**
1	54m29s	1	100%	1	25m59s	1	100%
2	30m59s	1.76	88%	2	13m15s	1.96	88%
4	16m3s	3.4	85%	4	6m52s	3.78	95%
8	9m3s	6.02	75%	8	4m46s	5.44	68%
16	5m58s	9.14	57%	16	3m55s	6.64	42%
32	4m35s	11.9	37%	32	2m44s	9.48	30%
64	4m18s	12.69	20%	64	2m37s	9.93	16%

Table 7.3: Speedup and efficiency of RDFS reasoner

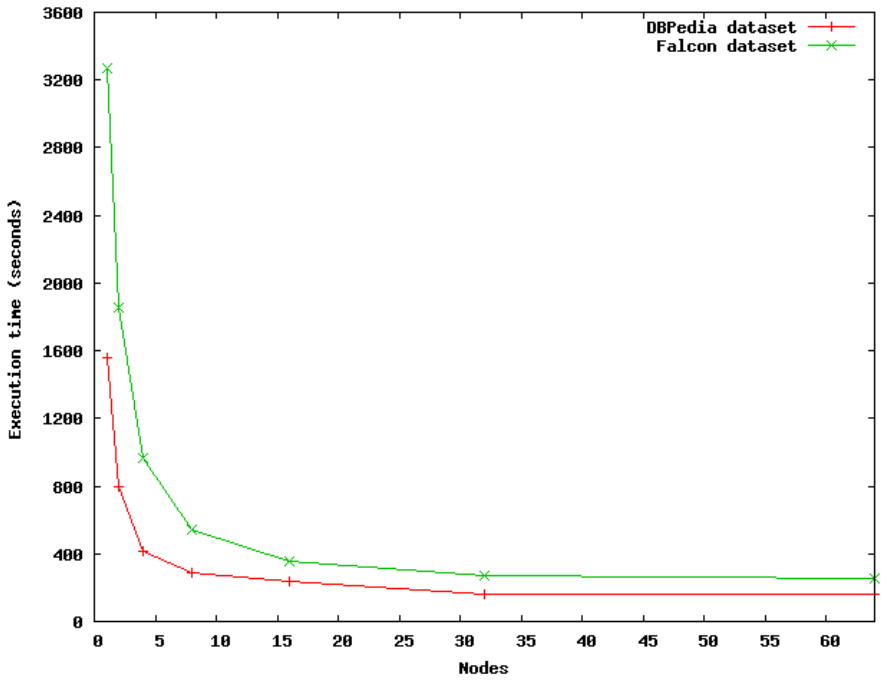

Figure 7.9: Scalability RDFS reasoner with different nodes

7.2.3 OWL reasoning performances

We have also tested the implementation of the OWL reasoning described in this paper but the performances are much worse than RDFS. We first present the performances using a real-world dataset of 35M of triples and then using a smaller and artificial data set of 6M of triples.

Performances on a real-world dataset

We first launched the reasoner on the Falcon dataset but the first trial did not succeeded. The jobs with the "sameAs" rules derived too many triples and we soon finished all the available space. We then tried to launch the reasoning excluding block 3 that is the one which applies the "sameAs" rules. The performances got better, but the program was still very slow and we stopped it after 12 hours, 130 jobs launched and more than 3.8B triples derived.

Performances on an artificial dataset

We tested the reasoner on a smaller and artificial dataset. We took the benchmark tool LUBM and we generated a small dataset of about 6 million triples. The reasoner finished the computation after 2h58m50s and derived about 5 million of new statements.

Looking at these results we conclude that the OWL implementation we have presented is very slow and inefficient, also compared with already existing reasoners.

An explanation could be that the ontologies we used contain data that do not comply to the standard and if we "blindly" apply the rules we just derive too many triples. However this conclusion is not enough to explain the poor performances since we have also tested the reasoner with a benchmark tool which generates a "clean" ontology. The main problem relies on the fact that we need to relaunch the jobs more than one time in order to derive everything. The optimizations we have introduced for the RDFS reasoner do not apply for some of the OWL rules and the performances are much worse than in the previous case.

Chapter 8

Future extensions and conclusions

In this thesis we address the problem of doing a scalable reasoning over a large amount of data. The problem is relevant in order to use the Semantic Web on a web-scale.

Our hypothesis is that we can design a scalable and efficient reasoning algorithm using a distributed approach, and, more specifically, using the MapReduce programming model.

In this work we have proposed some algorithms to verify our hypothesis and we tested the performances with different data sets and configurations. We have also implemented a dictionary encoding procedure to compress the input data and speed up the computation.

Initially we have discussed over a possible dummy implementation of the RDFS reasoning and we explained that there are too many problems with it. We have introduced three optimization in the algorithm that revealed to be crucial. These optimizations are:

- loading in memory the schema triples and executing the derivation on-the-fly;
- using a special rules execution order so that we avoid any loop;
- grouping the triples so that we eliminate duplicates during the derivation.

The results showed that the RDFS implementation is performant, computing the RDFS closure over the 1B triples from the 2008 Billion Triples challenge in less than one hour. We are not aware of any other implementation that neither can scale up to this size nor that can compute the closure in a time comparable to ours.

We have also designed an algorithm that does OWL reasoning but the results were disappointing. The algorithm is slow and fragile since we had to deactivate

some rules in order to execute it on real-world data sets. However the algorithm is still at a primitive stage and there are still many optimization to test.

After having analyzed the performances of the algorithms we conclude that our initial hypothesis is confirmed regarding RDFS reasoning. The performances we have measured show that our RDFS algorithm is efficient and scalable. Unfortunately we cannot conclude the same about OWL reasoning. Our approach showed to be slow and fragile. However, there are still too many optimizations to test and it is too early to give a definite answer.

The optimizations we can do on the OWL reasoning are left as a future work. We now mention some of them.

As first, we can optimize the rule's execution order. Unfortunately in OWL it is not possible to avoid the loops between the rules, but maybe it is possible to find an order for which the number of required jobs is minimal.

Another way to improve the poor OWL performances could be in finding a better way to execute the joins. Here, what we do is to apply the joins in a "naive" way, however we have shown for the RDFS reasoning that there are other ways, much faster.

One interesting extension could be to introduce some restrictions on the data in order to provide a more robust reasoning. In this thesis we "blindly" apply the rules without any data preprocessing but it was shown in [11] that the data in the Web contains many anomalies and that we need to filter out some dirty data that could lead either to an explosion of the derivation or to some contradictory statements. Such preprocessing could also prevent our reasoner from ontology hijacking that can happen when we crawl the data from the web.

Concluding, the work here presented has returned encouraging results. The OWL reasoner is not yet competitive but this is mainly due to its early development and it is too early to give a definite answer about its validity. On the other hand, the RDFS reasoner has shown good perfomances and scalability and we are not aware of any other approach with neither even comparable performances nor with a similar scalability. With future reearch we can refine the algorithms incrementing the performances and the quality of the derived information.

Bibliography

[1] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[2] D Brickley and RV Guha, editors. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, February 2004.

[3] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, pages 137–150, 2004.

[5] Stefan Decker, Sergey Melnik, Frank van Harmelen, Dieter Fensel, Michel C. A. Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4(5):63–74, 2000.

[6] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[7] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158 – 182, 2005. Selected Papers from the International Semantic Web Conference, 2004 - ISWC, 2004.

[8] Patrick Hayes, editor. *RDF Semantics*. W3C Recommendation, February 2004.

[9] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[10] J. Hendler and D. L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67–73, 2000.

[11] Aidan Hogan, Andreas Harth, and Axel Polleres. SAOR: Authoritative reasoning for the web. In *Asian Semantic Web Conference*, pages 76–90, 2008.

[12] P. N. Johnson-Laird. Deductive reasoning. *Annual Review of Psychology*, 50(1):109–135, 1999.

[13] Atanas Kiryakov et al. *Validation goals and metrics for the LarKC platform.* LarKC deliverable, July 2009.

[14] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlim - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.

[15] Graham Klyne and Jeremy J. Carroll, editors. *Resource Description Framework: Concepts and Abstract Syntax.* W3C Recommendation, February 2004.

[16] Ralf Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[17] Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a complete OWL ontology benchmark. In *European Semantic Web Conference*, pages 125–139, 2006.

[18] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[19] Peter Mika and Giovanni Tummarello. Web semantics in the clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[21] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronald Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: A platform for large-scale analysis of Semantic Web data. In *Proceedings of the International Web Science conference*, March 2009.

[22] David Robertson, Grigoris Antoniou, and Frank van Harmelen. *A Semantic Web Primer.* The MIT Press, 2004.

[23] Anne Schlicht and Heiner Stuckenschmidt. Distributed resolution for ALC. In *Proceedings of the International Workshop on Description Logics*, 2008.

[24] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In *International Semantic Web Conference*, pages 82–97, 2008.

[25] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[26] Ramakrishna Soma and V. K. Prasanna. Parallel inferencing for OWL knowledge bases. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 75–82, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Herman 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.

[28] Ora Lassila Tim Berners-Lee, James A. Hendler. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[29] Dmitry Tsarkov and Ian Horrocks. Description logic reasoner: System description. In *IJCAR*, pages 292–297, 2006.

[30] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Mapreduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[31] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.