University of Zurich[UZH]

Distributed RDF Reasoning and Graph-Based Approaches

Thesis July 7, 2016

Felix Kieber
of Schaan, Liechtenstein

Student-ID: 08-731-945
felix.kieber@uzh.ch

Advisor: **Shen Gao**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank the University of Zürich, the Department of Informatics, and the Dynamics and Distributed Information Systems Group and its head Professor Abraham Bernstein for providing the environment and infrastructure that allowed me to write this Bachelor thesis.

Furthermore, I thank Shen Gao, which functioned as advisor for this thesis and supported me throughout the process of putting it together.

Zusammenfassung

Diese Bachelorarbeit rekapituliert bestehende Ansätze zu verteiltem, large-scale RDF Reasoning, basierend auf dem MapReduce-Konzept. Im speziellen wird die bestehende Inferenz-Engine *Cichlid* genauer untersucht und es werden einige Verbesserungen vorgeschlagen. Daraufhin wird ein graphenbasiertes Konzept für RDF Reasoning vorgestellt mit konkreten Implementierungsbeispielen. Im besonderen beinhaltet die Arbeit auch einen alternativen Ansatz zur Anwendung von transitiven Inferenzregeln. Mittels eines Pregel-basierten Algorithmus wird dazu die transitive Hülle des RDF Graphen berechnet. Tests zeigen schliesslich die grundsätzliche Funktionalität der graphenbasierten Ansätze auf. Konkrete Werte für anwendungsnahen Einsatz sowie Vergleichswerte zu bestehenden Methoden sind nur von begrenzter Aussagekraft.

Abstract

This Bachelor thesis recapitulates existing approaches towards distributed, large-scale RDF reasoning, which are based on the MapReduce model. Specifically, the existing inference engine *Cichlid* will be analyzed more closes and some improvements are suggested. Following this, a graph-based approach towards RDF reasoning will be presented, along with concrete examples for implementation. In particular, this thesis includes an alternate method for applying transitive inference rules. For this, a Pregel-based algorithm computes the transitive closure of the RDF graph. Tests show the functionality of the graph-based approaches. Concrete measurements of real-world performance and comparison to existing approaches are of limited meaningfulness.

Table of Contents

1

Introduction

Along with the increased demand for big data solutions in the business world in recent years, various new tools and approaches have been developed to cope with the ever-increasing scale of problems faced. While most technologies are still relatively young, they are already being applied to a wide range of problem fields.

One very obvious area of research which can benefit of highly scalable approaches is the World Wide Web, traditionally an area characterized by large amounts of data. It is thus no big surprise that multiple technologies which are now being used were originally developed at Google [Dean and Ghemawat, 2004] [Malewicz et al., 2010].

As an extension to the existing World Wide Web, the *W3C* introduced various technologies related to the *Semantic Web*. This is an effort towards enriching the existing Web of documents with a *Web of data*.

Of particular interest to this thesis is the area of RDF reasoning within the Semantic Web stack. The problem of RDF reasoning combines the scales of big data with discovery of new information by applying logic rules. As such, there already exists research which approaches the task with the help of recent big data technologies, such as MapReduce and Spark [Urbani et al., 2012] [Gu et al., 2015].

In addition to analyzing these existing approaches and suggesting some improvements over the inference engine *Cichlid*, this thesis also examines an alternate approach towards RDF reasoning, by employing graph-based techniques.

Thus, the next chapter will cover core concepts and technologies related to the Semantic Web and RDF. Then follows the analysis of existing approaches, together with suggestions for possible improvements. After that, the graph based paradigm will be introduced and examined, and some limited tests will be conducted. The thesis will be concluded by examining the limitations of this work, possible next steps and conclusions, that can be drawn.

2

The Semantic Web

The *Semantic Web* is a set of standards provided by the W3C intended as an extension to the existing Web, primarily a Web of documents, to enable a *Web of linked data*. Thus, these standards form a common framework which allows data to be shared across the web and across applications, by defining common formats for creating and describing data, building vocabularies and rules for handling data. These efforts resulted in various technologies, such as RDF, RDFS, OWL and SPARQL.[1]

At its core, the *Semantic Web* is a Web of data, that is, a large collection of linked data across the Web. This requires, for one, the data itself as well as relationships which link data and form a web. The *Resource Description Framework*, RDF, is the foundation for presenting data on the web in a common format. It defines an abstract syntax which serves as language for data on the Web. The following sections will present RDF and related technologies, as a good grasp of these is vital to understanding the work presented in this thesis.

2.1 RDF

RDF defines a data model for representing information about data on the Web. It has been formally specified by a collection of documents from the W3C.[2] Its core structure is an RDF-Graph, formed by a set of triples. These triples consist of subject, predicate and object, where subject and object are nodes of the graph, and the triple itself a directed edge, connecting both nodes.

Nodes are either an IRI (Internationalized Resource Identifier), a literal or a blank node. IRIs and literals denote, or refer to entities, which intuitively can be any *thing*, such as objects in the real world, concepts or numbers. These entities are called resources.

A triple then states that there is a relationship between subject and object, indicated by the predicate. The triple *Tolkien - isAuthorOf - The Hobbit*, for instance, can be read as *Tolkien is the author of The Hobbit*. This is known as an *RDF Statement*.

An important aspect of RDF is that predicates are IRIs as well and can, as we will see below, appear in the position of a subject or object within a triple.

[1] *https://www.w3.org/standards/semanticweb/*
[2] *https://www.w3.org/standards/techs/rdf#w3c_all*

There are further intricacies to the RDF Standard, such as differences between IRI, literal and blank nodes or namespaces and prefixes. They are, however, beyond the discussion of this thesis.

2.1.1 RDF Serialization Formats

It is worth noting that RDF, as described above, only defines an abstract syntax and not a specific format of writing RDF data. Consequently, various concrete syntaxes have been developed to provide common and well-known formats, most of which are specified by the W3C.

The first standard developed by the W3C is *RDF/XML*, which as the name implies is an XML syntax for RDF. This explicit and verbose syntax is however not ideal for human readability. Luckily, other, more practical and compact serialization formats have been developed, which simplify writing and reading RDF documents. Of particular relevance to this thesis is the *N-Triples Language*.[3] Arguably one of the simplest languages, it defines triples per-line as a sequence of subject, predicate and object, separated by spaces or tabs and terminated by `.` and newline. Triples could then look as follows:

```
Tolkien isAuthorOf TheHobbit .
Rowling isAuthorOf HarryPotter .
```

This sequential format is very suitable for the purposes of Big Data Processing, where large text files are often read in a streaming manner, line by line. As we will see later, when RDF data is given in this format, very little preprocessing is required to transform the lines to triples.

2.2 RDFS

RDF on its own does not specify what IRIs denote. Conversely, there is no rule, which specifies how an IRI that refers to a particular resource should look like. Thus, a concrete dataset usually uses a vocabulary, a collection of IRIs to be used in the RDF graph.

As an additional tool in specifying vocabularies, the W3C provides *RDF Schema*, or *RDFS*, as a set of vocabularies.[4] The meaning of the IRIs it contains is well defined and allows for data-modeling, i.e., specifying well known relations between resources in a given vocabulary. As such, it is a semantic extension of RDF.

More specifically, RDFS introduces the concept of classes, a concept well known from Object Oriented Programming. This allows the grouping of IRIs of the same kind. In this context, a class is defined by the set of instances to which it applies. To avoid confusion one should note the distinction between a class and an instance of that class, as both are presented as IRIs. The most important IRIs defined by RDFS can be seen in Table 2.1.

[3] *https://www.w3.org/TR/2014/REC-n-triples-20140225/*
[4] *https://www.w3.org/TR/2014/REC-rdf-schema-20140225/*

IRI	Meaning
Classes	
rdfs:Resource	The class of *everything* described by RDF. All IRIs describe instances of this class, and all classes are subclasses of rdfs:Resource.
rdfs:Class	The class of classes. Every class in an RDF dataset is an instance of rdfs:Class.
rdfs:Property	The class of properties, those IRIs that appear as predicates in triples.
Properties	
rdfs:range	A property which states when given the triple p *rdfs:range* c that the *object* of a triple with predicate p belongs to class c.
rdfs:domain	Analogous to rdfs:range, rdfs:domain states that the *subject* of a triple with predicate p belongs to c.
rdf:type	A property that states when i *rdf:type* c, that i is an instance of class c.
rdfs:subClassOf	A property that allows to state a subclass relation between two classes. This is a transitive property.
rdfs:subPropertyOf	A property roughly equivalent to rdfs:subClassOf, allows to state this transitive relation between two properties.

Table 2.1: RDFS Vocabulary Excerpt

The properties defined by RDFS are of special interest to us, as will be seen below when discussing Reasoning and Inference Rules.

2.3 OWL

The *Web Ontology Language OWL*[5], will mostly be left outside the scope of this thesis. It nevertheless is a core technology in the *Semantic Web* stack. As such, it warrants a brief mentioning. OWL is another ontology language for RDF, much in the same vein as RDFS. It introduces another set of vocabularies for defining ontologies building upon the RDFS vocabulary. The difference between vocabularies and ontologies is not clear cut, but generally, ontologies are considered to be more formal and extensive.

OWL defines further relations between entities, such as equality and inequality (*owl:sameAs, owl:differentFrom*), or complex class constructs, built from the intersection, union or complement of other classes. It also extends properties by defining various property characteristics, such as symmetric, asymmetric, reflexive or transitive, as well as relations between properties (*owl:inverseOf, owl:propertyDisjointWith*). It is evident, that

[5] *https://www.w3.org/TR/2012/REC-owl2-primer-20121211/*

Rule #	Condition	Consequence
rdfs2	s p o & p rdfs:domain c	s rdf:type c
rdfs3	s p o & p rdfs:range c	o rdf:type c
rdfs5	a rdfs:subPropertyOf b & b rdfs:subPropertyOf c	a rdfs:subPropertyOf c
rdfs7	s p o & p rdfs:subPropertyOf q	s q o
rdfs9	a rdf:type x & x rdfs:subClassOf y	a rdf:type y
rdfs11	a rdfs:subClassOf b & b rdfs:subClassOf c	a rdfs:subClassOf c

Table 2.2: RDFS Entailment Patterns with Two Antecedents

OWL allows much more intricate ontologies to be modeled.

2.4 Reasoning

The *W3C* produced various documents specifying the formal aspects of RDF, RDFS and OWL Semantics. Approaching these formalities would easily warrant its own thesis. Relevant for the task of reasoning, which concerns itself with the discovery of new relations within existing structures, are RDF inference rules. These allow new triples to be inferred by combining existing triples with specific patterns.

Given the well defined meaning of the RDF and RDFS vocabulary, certain inference rules can be formulated. These rules have a condition or *antecedents*, from which a potentially new triple can be inferred (See Table 2.2 for the RDFS entailment rules with two antecedent triples). OWL also introduces further inference rules. Specifically, the Ter Horst ruleset forms a compromise between computational feasibility and usefulness [Ter Horst, 2005].

A common task is to exhaustively apply the inference rules on to a given RDF dataset until no new triples can be derived. This is called *closure computation*, the main task this thesis is concerned with. As an illustration of RDF reasoning, consider the following example: Say, we know that an *Audi* is a specific kind of, or *subclass* of, *car*. We also know that a *car* is a specific kind of *vehicle*. In a concrete RDF dataset, this could be represented as the two triples

```
Audi rdfs:subClassOf Car .
Car rdfs:subClassOf Vehicle .
```

Intuitively, following an everyday understanding of these terms, we are able to infer from this that *Audi* is specific kind of *vehicle*, i.e. the triple

```
Audi rdfs:subClassOf Vehicle .
```

And in fact, this is what the rule rdfs11 describes in a more formal way.

3

Reasoning With The MapReduce Model

Given the potentially enormous size of RDF datasets, the amount of processing required for the reasoning task quickly exceeds the capabilities of even very powerful single machines and makes traditional computation approaches increasingly infeasible. Hence, the field of Big Data becomes of interest to the task at hand. Especially considering the possibility of even larger datasets in the future.

Originally a problem which pioneering Web-Startups such as Google and Facebook had to face, more and more traditional companies encounter the question of how to handle the ever-increasing amount of data at their disposal; data such as customer records, bills of material or the whole World Wide Web.

A key idea for processing large datasets is *parallelization* - dividing the task not only between multiple cores, but between multiple machines within a distributed system. In the next section 3.1, we will have a look at the well established MapReduce approach.

There is already some previous work towards a distributed solution to apply inference rules to RDF datasets. Most notably [Urbani et al., 2012], presenting a MapReduce framework for RDF inference and [Gu et al., 2015], which develops on the previous paper and implements the concept with Spark (See 3.1).

This chapter will mainly build on the mentioned contributions, analyzing them and improving on *Cichlid*, a inference engine built on the Spark framework.

3.1 MapReduce and Related Technologies

Since its introduction, MapReduce has been widely adopted as an approach to Big Data processing. While the idea of distributing a task among multiple workers has been around before, implementations usually were designed for specific tasks. MapReduce thus is an important contribution towards more generic Big Data processing [Dean and Ghemawat, 2004].

The base concept of MapReduce is quite straightforward to understand, yet it is applicable to a wide range of problems. The input format of MapReduce is key/value pairs. The computation itself consists of a *map* and a *reduce* phase. These two phases are implemented by user-supplied map and reduce functions.

Conceptually, first the map function is applied to every input pair individually. Taking the pair, the function then produces as output a set of key/value pairs - this set can contain an arbitrary amount of entries: 0, 1 or more.

Listing 3.1: A simple MapReduce word count example

```
1  map(int key, string value) {
2      array[string] words := value.split(" ")
3      for word in words {
4          emit_intermediate(word, 1)
5      }
6  }
7
8  reduce(string key, iterator[string] values) {
9      int count = 0
10     for value in values {
11         ++count
12     }
13     emit(key + " " + count)
14 }
```

The key/value pairs generated in this way are then grouped by key, and passed to the reduce function. This function is applied to every key, together with the list of values that have the respective key. From this, the reduce function produces some output.

This high-level view of MapReduce does not make the parallelization explicit. A programmer could arguably write a valid MapReduce program without having to concern themselves with the distributed side of the program. Nevertheless, it is advisable to understand what is going on under the surface of the simple API, especially to understand what kind of tasks are suitable for MapReduce and which tasks are not.

Parallelization, i.e. workload distribution is achieved by splitting the input - commonly a large text file available on a (distributed) file system - into so called *input splits*. These splits are distributed among worker nodes within a cluster, which individually process their split(s) according to the provided map function. The reduce step again is assigned to multiple worker nodes. This step usually incurs some network traffic, as the reduce-nodes have to collect the values from potentially multiple map-nodes. Commonly, each reduce-node produces its own final output.

Listing 3.1 shows a canonical example of a MapReduce program to count the number of occurrences of each word in a text. The input would be a text file, each line is an input pair of the form `(lineNumber, line)`. In this example, line numbers are not required. Each line, in the form of a string, is split (naively) into individual words, and every word is emitted, with the word as key and the number 1 as value.

The reduce function then receives a word as key and an iterator which contains n 1s, where n is the number of occurrences of the word in the text. The reducer then only has to sum up the iterator. In our simple case, the iterator only contains 1s, thus the size of the iterator is the individual word count.

A interesting and useful characteristic of MapReduce is that standard *Relational Al-*

gebra operations, commonly used in SQL, can be modeled with MapReduce. Namely: *Selection, Projection, Union, Intersection, Difference, Natural Join, Grouping and Aggregation* [Leskovec et al., 2014, p. 32ff].

3.1.1 Apache Hadoop

The Apache open-source project *Hadoop* is a popular platform for Big Data, mainly using the Java programming language.[1] It includes the *Hadoop Distributed File System, HDFS*, a cluster resource management and job scheduling framework called *YARN* and an implementation of MapReduce (*Hadoop MapReduce*), based on YARN. Hadoop also spawned various related projects, which build upon the platform and extend it with further functionality, such as distributed databases (*HBase*[2]) or data serialization (*Avro*[3]).

3.1.2 Apache Spark

Apache Spark[4] is another project related to Hadoop. In many ways it can be seen as a development upon MapReduce, retaining its properties such as scalability and fault tolerance, while also supporting other types of applications which MapReduce traditionally is not well suited for. Specifically, iterative algorithms which repeatedly run computations over a dataset and interactive analysis of datasets [Zaharia et al., 2010].

Spark introduces *resilient distributed datasets, RDDs*, immutable, i.e. read-only collections of objects which are partitioned across a cluster. The RDD API provides an array of operations on RDDs, which are separated into *transformations* and *actions*. Transformations are generally operations which result in a new RDD. They are also *lazily evaluated*, which means they are only computed when their result is needed. This is triggered by actions, which result in a value.

Listing 3.2 shows a simple usage of the Spark API. The `SparkContext` object `sc`, assumed to have been previously instantiated, provides a method for loading text files into an `RDD[String]`, where each string object is a line of the input text file. It then filters out those lines which contain the substring `filterword`, returning an RDD containing the remaining lines. Finally, `.count()` returns the numbers of entries in the RDD.

Note that `.count()` is an action as described above and triggers the evaluation of `.textFile(...)` and `.filter(...)`. Lines 6-8 show the same operations, chained together as one expression. Lines 3 and 7 contain two different shorthand notations for lambda functions in Scala; in this case, the functions take a line as parameter and return false or true, depending on whether the line contains the substring `filterword` or not, respectively.

Spark is mainly written in the JVM programming language Scala and provides APIs for Scala, Java and Python. While working with Spark, using the Scala API proved to be

[1] *http://hadoop.apache.org/*
[2] *http://hbase.apache.org/*
[3] *http://avro.apache.org/*
[4] *http://spark.apache.org/*

Listing 3.2: A simple Spark example in Scala

```
1 val textFile: RDD[String] = sc.textFile("inputfile.txt")
2 val filteredText = textFile.filter(
3     line => !line.contains("filterword"))
4 val filteredLineCount = filteredText.count()
5 // This could also be chained
6 val flc2 = textFile
7     .filter( !_.contains("filterword") )
8     .count()
```

the most productive and pleasant experience when writing Spark applications. However, this assessment is subjective, and there is not much difference in the capabilities of the APIs. There are however some modules of Spark which do, as of this writing, not provide a Java or Python API.

3.2 WebPIE and Cichlid

The *WebPIE inference engine* is the result of extensive research in the area of web-scale RDF inference [Urbani et al., 2012] using MapReduce. It addresses many principal issues faced when conducting large scale reasoning in parallel, such as joining of large datasets, handling of duplicates and optimized rule execution order (see Subsection 3.3 and Figure 3.1). Especially for RDFS entailment rules a very feasible model has been developed.

The *Cichild* engine then takes these concepts and implements them in Apache Spark [Gu et al., 2015]. The model presented for WebPIE requires the iterative execution of multiple MapReduce jobs. An individual MapReduce job reads data from, and writes it back to disk storage, which makes sequential or iterative perform a lot of time consuming writing and reading. Spark specifically improves upon this by loading the data into memory and keeping it there as long as required or possible. This improves the performance of chained MapReduce operations and makes the Spark framework well suited for the task.

3.3 Rule Application Approaches

The general approach towards inference is taking an inference rule and applying it to the dataset. When considering only the RDFS entailment rules, as a first step the rules can be split in two groups: Those with only one condition or antecedent, and those with two. No RDFS inference rule has more than two antecedents, unlike the OWL rule set.

Application of a one-antecedent rule is trivial and can be done within a single iteration over the data. As such, the task is easily parallelized. In fact, all the single-antecedent rules can be applied within one traversal. Listing 3.3 shows an example implementation,

applying the rules 4a, 4b, 6 and 10. Furthermore, it is debatable whether the new triples inferred from these rules are of any practical value. They nevertheless may serve as a good starting point for analyzing the problems that arise with this approach to RDFS reasoning.

Duplicate Output Triples

This direct approach may however yield a lot of duplicates. Consider the two triples `(s,p1,o1)` and `(s,p2,o2)`. The triple `(s, rdfs:type, rdfs:Resource)` would be inferred twice. This is even more apparent if we took n triples, that only differed in their predicate. Of $2n$ inferred triples, all but two would be duplicates. Each input triple yields either two or three output triples (notice that rule 6 and 10 exclude each other). Thus, the number of output triples s for n input triples is $2n \leq s \leq 3n$.

A more space efficient approach for rules 4a and 4b would thus be to collect all distinct subjects and objects in the dataset and then map all resources to the resulting rule (See Listing 3.4).

Iterative Application

Notice furthermore, that the previously discussed rules only need to be applied once to the dataset, to infer all possible triples. Rules 4a and 4b apply to an arbitrary triple `(x,p,o)` or `(s,p,x)`, and return a triple of the form `(x, rdfs:type, rdfs:Resource)`. For any such output triple, rule 4a would yield the identical triple, and rule 4b would yield the triple `(rdfs:Resource, rdf:type, rdfs:Resource)`, which for completeness sake can be added manually.

There are in fact RDFS entailment rules, that yield new triples on which the same rule can be applied again and may yield again new triples. These rules have to be applied iteratively until no new triples can be inferred. This leads us to the second group of RDFS rules according to the distinction made at the beginning of this section.

RDFS Rules with Two Antecedents

Rules 5 and 11 both need to be applied iteratively. Intuitively, this makes sense: If we know that A is a subclass of B, and B is a subclass of C, we can infer that A is a subclass of C. Now, if we also know that C is a subclass of D, we can infer that A is a subclass of D, but only in the next iteration.

These two rules can be classified as *transitive* rules. In fact, exhaustively applying rules 5 and 11 is equivalent to calculating the transitive closure of the RDF graph containing only the nodes that are connected via the rdfs:subPropertyOf and rdfs:subClassOf predicate respectively [Gu et al., 2015].

By filtering the input dataset to only contain the triples with the subProperty and subClass predicate respectively, and then remove the now-redundant predicate, the resulting presentation of the data is suitable for the application of *smart transitive closure* as presented in [Leskovec et al., 2014].

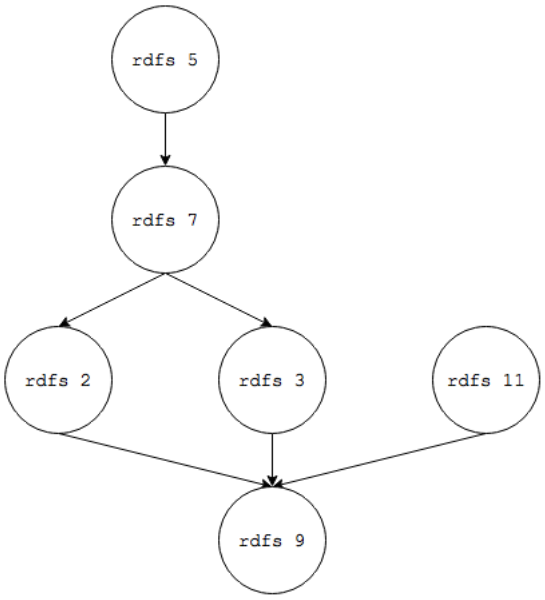

Figure 3.1: A cycle-free sequence graph for RDFS rules with two antecedents.

The question of whether a rule has to be applied multiple times not only occurs on the level of a single rule. It must also be considered, whether the output of one rule may serve as input to another rule and produce new triples. Figure 3.1 shows the dependencies between the RDFS entailment rules in a graph, ignoring rare edge cases. This directly translates to the sequence in which rules have to be applied in order to avoid multiple iterations [Urbani et al., 2012].

Another issue for rules with two antecedents is that two datasets have to be joined to apply the rule. A naive approach would be to self-join the data. However, for large datasets this may be a very expensive operation. Luckily, this can be circumvented. The input RDF set can be split into *instance* triples and *schema* triples. The schema set contains all triples with an *rdfs:** predicate and the instance set all the other triples. Furthermore, the schema set is very small compared to the instance set. Another observation is that for all rules in question, one or both antecedents are from the schema set. Hence, the relevant schema information can be broadcast across the cluster, i.e. made available locally on each node. This prevents having to join two large datasets across the cluster [Urbani et al., 2012] [Gu et al., 2015].

3.4 Improvements to Cichlid

As the first part of the practical work done, the *Cichlid* Engine introduced in 3.2 will be reviewed and modified in places. With a lot of theoretical groundwork already laid down in [Urbani et al., 2012], a Spark implementation can adopt these optimizations. As such, building a Spark version is mainly the task of porting existing concepts and

Listing 3.3: Single-antecedent Rule Application

```
val triples: RDD[(String, String, String)] = ...
val newTriples = triples.flatMap{
    case (s, p, o) => {
        var lst = List(
            (s, "rdfs:type", "rdfs:Resource"),
            (o, "rdfs:type", "rdfs:Resource")
        )
        if (p.equals("rdf:type")) {
            if (o.equals("rdf:Property"))
                lst = lst + (s, "rdfs:subPropertyOf", s)
            else if (o.equals("rdfs:Class"))
                lst = lst + (s, "rdfs:subClassOf", s)
        }
        lst
    }
    .distinct()
}
```

Listing 3.4: Modified Rule 4a and 4b

```
val triples: RDD[(String, String, String)] = ...
val resources = triples.flatMap{
    case (s,p,o) => {
        if (s.equals(o))
            List(s)
        else
            List(s, o)
    }
    .distinct()
val rule4ab = resources.map(
    (_, "rdf:type", "rdfs:Resource"))
}
```

the main challenges lies in optimally employing Scala and the Spark framework.

Looking at the openly available source code for Cichlid[5], the following issues were identified and addressed.

3.4.1 Discarded SubClass and SubProperty Schema Triples

The RDFS reasoning program contains a logical error. It does not cause any error, however schema triples could be silently discarded an hence certain possible triples may not be inferred as a consequence.

Listing 3.5: Cichlid implementation silently discards triples with map-conversion

```
1 val subprop, subclass, domain, range: Array[(String, String)]
2
3 val spClosure = transitiveClosure(subprop.toSet)
4 val scClosure = transitiveClosure(subclass.toSet)
5
6 val sp = sc.broadcast(spClosure.toMap)   // ISSUE
7 val cl = sc.broadcast(scClosure.toMap)   // ISSUE
8 val dm = sc.broadcast(domain.toMap)      // ISSUE
9 val rg = sc.broadcast(range.toMap)       // ISSUE
```

Listing 3.5 shows an excerpt from the original implementation in Cichlid, where the transitive RDFS rules 5 and 11 are applied and all schema-maps are broadcast to the nodes in the cluster. It has been formatted for easier understanding. The objects `subprop, subclass, domain` and `range` from the first line contain all existing schema triples with the according predicate. In addition, the predicate has been removed from the triples, leaving only *subject-object* tuples.

The transitive closure is applied on lines 3 & 4, returning a set containing all inferable triples. The concrete implementation of the function `transitiveClosure(...)` is not important here and works as intended. The issue arises in lines 6 & 9. By transforming a set of tuples into a map, entries may be lost. Consider the tuples *(a,b)* and *(a,c)* in a set. Converting this set using the `.toMap()` method provided by Scala, one of the tuples is lost as a becomes the key, and keys must be unique. In fact, for n tuples with the same subject, $n-1$ are lost.

Luckily, this problem is simple to solve. Two solutions are possible: either the closures are kept as sets of tuples, or a custom *toMap* function is implemented, that maps the subjects to a set of all objects a given subject maps to. The first approach has the drawback that a set does not directly provide key lookup. Later rule applications would have to iterate over the whole set of tuples, collecting entries where the first element is the key in question. Thus, the second approach was chosen by providing the implementation shown in Listing 3.6.

When tested with an example dataset of approximately 1.3 million triples this error made almost no difference. Only a single additional triple was inferred with the corrected

[5] *https://github.com/PasaLab/cichlid*

Listing 3.6: Alternative toMap-implementation that avoids dropping tuples

```
1 def setToGroupedMap[A, B](s: Set[(A, B)]): Map[A, Set[B]] =
2     s.groupBy(t => t._1)
3         .map {
4             case (k, vset) => (k, vset.map( t => t._2 ))
5         }
```

implementation. Other rules may also compensate for the error, by producing triples that were lost by the error. At any rate, the impact of this error depends on the completeness of the existing schema and constellation of the instance triples.

Additionally, changing the broadcast objects from `Map[String, String]` to `Map[String, Set[String]]` naturally requires some adjustment in the operations utilizing these values. Specifically, this applies to rules 2, 3, 7 and 9. Listing 3.7 shows how this may look for rule 7. The change is done on lines 3 - 5. Instead of a simple map, yielding always a single triple, a *flatMap* is applied which maps to a sequence of one or more elements (triples with a predicate that is not present in the subProperty-map are filtered out on line 2, so no empty sequences will be yielded).

Listing 3.7: Adapted application of RDFS rule 7

```
1 val r7_out = triples
2     .filter(t => sp.value.contains(t._2))
3     .flatMap( t => {
4         for (p <- sp.value(t._2)) yield (t._1, p, t._3)
5     })
6     .union(triples)
7     .distinct()
```

3.4.2 Improving Elimination of Duplicates

Applying the rules to every triple in the dataset yield a large amount of output triples, many of which are duplicates. Without addressing the duplicates at all, the output size quickly grows far too large to handle, even in distributed clusters. However, by preprocessing the input set accordingly, the amount of duplicates generated can be reduced and it can be ensured that the remaining duplicates are generated on the same node. Thus, they can be easily eliminated without the need for further shuffles [Urbani et al., 2012].

However, Cichlid does not fully implement this preprocessing. Instead it filters the input set to only contain those triples, where the predicate is also in the according schema map and then removes all duplicates at the end. Therefore, an alternate implementation has been devised which handles duplicates according to the approach described above, an example of which can be seen in Listing 3.8.

The modification is twofold; The first change happens on lines 2 & 3. Recall RDFS rule 3, which states:

$$p\ range\ c \quad \& \quad x\ p\ i \quad \Rightarrow \quad i\ type\ c$$

Notice how the subject x in the input triple is irrelevant to the output. For a given schema triple, all instance triples that only differ in their subject produce the same output when rule 3 is applied. Thus, the triples are mapped to tuples containing only predicate and object (line 3), and duplicate tuples are removed from this RDD (line 4).

Next, the tuples are grouped by their object (line 5). With this, all tuples with the same object can be processed together. Between lines 6 to 9, all grouped tuples are joined with their entries in the broadcast schema map. Because the tuples were grouped by their object, all output tuples (with the implicit predicate rdf:type) containing that object (now as subject), are ensured to be on the same node. Duplicates can thus easily be removed by converting the resulting collection to a set.

The same concept can be applied analogously to rule 2. Rules 7 and 9 also allow for grouping, but as they require both subject and object of the input triples, no initial duplicate removal can be done.

Listing 3.8: Modified application of RDFS rule 3 that locally handles duplicates

```
1  val r3_local = triples
2      .filter(t => rg.value.contains(t._2))
3      .map( t => (t._2, t._3) )
4      .distinct()
5      .groupBy(t => t._2)
6      .flatMap(grp => {
7          (for ((p,o) <- grp._2; c <- rg.value(p))
8              yield (o, c)
9          ).toSet
10     })
```

Tests show that this implementation performs slower than the original approach. This makes sense, considering that the removal of duplicate input tuples on line 4 requires a shuffle. In the same way, the grouping operation is also costly. However, as described above, this implementation minimizes the amount of duplicates and allows the remaining duplicates to be removed locally.

4

Graph-Based Approach to Reasoning

As seen in 2.1, the abstract syntax of RDF describes a directed graph, so RDF data lends itself to also be modeled as a graph in memory. Various graph databases, introduced as early as the eighties since have been pushed aside by other database models. With the increase of problems that are suited for network and graph modeling, such databases have become relevant again [Angles and Gutierrez, 2008].

Previous research considering RDF and graph databases has focused on RDF querying [Angles and Gutierrez, 2005]. This part of the thesis will examine whether there are useful approaches towards RDF reasoning based on the graph representation.

To this end, first the *Spark* graph processing module *GraphX* will be introduced. Then we examine how we can concretely model the RDF data as a graph to suit our requirements. Finally, two approaches are introduced to compute the application of specific rules.

4.1 Pregel and Spark GraphX

Pregel is a computational model designed for processing large-scale graphs with millions to billions of vertices [Malewicz et al., 2010]. It views every vertex as a computational unit that can send messages along its edges to neighboring vertices, and compute new values and messages based on the messages it receives.

The execution is structured into supersteps. In each step, every vertex receives zero or more messages, computes a new value based on its existing value and the messages received and finally, optionally computes and sends a new message along its edges. A vertex can chose to become inactive and is woken up when it receives a messages. When all vertices are inactive, the process ends.

Apache Spark offers a module for big data graph processing called *GraphX*. This module contains a `Graph` structure built on Spark RDDs. More specifically a GraphX `Graph` consists of a `VertexRDD` and a `EdgeRDD`. Vertices have a unique Long-id and an attribute, which can be of an arbitrary type. Edges also have a generic attribute, as well as the id of the two vertices it connects. As edges are directed in GraphX, one id is the source and one is the destination. Additionally, multiple edges from a vertex A to vertex B are allowed. This makes the GraphX graph a *directed multi-graph* with edge attributes.

GraphX provides various methods on the Graph-structure to allow filtering, mapping or message aggregation, as well as easy access to vertex- and edge-RDD. Additionally, it exposes a variant of the Pregel API.

4.2 RDF Datasets as a Graph

The RDF data model is already a graph, thus no work is required to translate it to that format. However, to avoid confusion, some terminology and characteristics of the RDF graph should be made explicit.

Consider the RDF triple *(s p o)*. *s* and *o* map to vertices, connected by an edge. Notice however, that *p* alone does not comprise this edge, as it lacks the information on which two nodes it connects. The edge is formed by the *whole triple*. Thus, *p* alone can be interpreted as the *attribute* of that edge. In short, we have:

vertex subject, object

edge triple

edge attribute predicate

However, another complicating factor arises from the fact that subjects and objects, *as well as* predicates are resources. In other words, *p* from the above example could appear in a second triple *(p rdfs:range c)*, where it is the subject and would be translated to a node in the graph. This means, that properties can map to nodes or edge attributes, depending on their function, i.e. position in the triple.

To clarify, property should not be confused with predicate. Predicate describes the function and position of a resource in the triple. Property, more precisely rdfs:Property, is the class of all resources, that appear as predicates. So every predicate is a property, but not every occurrence of a property in a triple is necessarily as a predicate.

Ultimately, a consistent graph can be built by adhering to the mapping mentioned above which only considers the position of resources. This also means that information on individual properties is connected to a singular node representing that property, however the connecting function of a given property as predicate is distributed across the graph as edge properties. Subsection 4.2.3 will show how this is not ideal.

4.2.1 Loading an RDF Graph

While providing useful methods in terms of graph processing once the graph is loaded, GraphX does, as of this writing, not offer any helpful utilities to build a graph from our input data. With vertices and edges present in the appropriate format a graph can be created, but transforming the input data to the required format has to be done manually and presents a challenge in itself.*N-Triples* is used as input format.

The difficulty is that GraphX vertices are identified by a `VertexId`, which is an alias for `Long`. The edges then are defined by the `VertexId`s of the vertices it connects, as well

as an edge attribute. Thus, we need all distinct vertices and assign a unique id to them. Furthermore we must read the triples, replace subject and object by their respective unique id and set the predicate as the attribute of that edge.

This is not too challenging when done locally, sequentially reading the input file. Each line can be split into subject, object and predicate. For the former two, i.e. the nodes, a map can be used to keep track of nodes already seen before. If a node has not been seen before, it is assigned a unique id by incrementing a local counter variable and then stored in the aforementioned map. This way, the edges can be built at the same time, by either using the id stored in the map if the node has been encountered before, or assigning it a new id (and storing it in the map).

However, this approach requires the complete lookup map for node ids to be available locally while also relying on a single local counter to assign the ids. Seeing as this thesis specifically examines RDF reasoning at the scale of big data, it cannot be assumed that the input data fits into the local memory of a single machine.

Thus, a distributed approach of loading the data is required. Two implementations have been devised which utilize the Spark framework. Listing 4.1 shows the initial implementation. The RDD `nodes` holds the tuples of all distinct nodes and their assigned unique id. Note that `.zipWithUniqueId()`(line 6) does not guarantee continuous ids, this is however not required.

Between lines 8 and 15, the `triples` RDD is joined with nodes by subject. This allows the name of the subject (its IRI) to be replaced by its assigned id. The resulting RDD is then joined with nodes analogously, this time by object. This way, subject and object in all triples can be replaced by their id.

This approach, while functioning, is very slow. It requires multiple grouping operations and joins over the whole triple set, which is very costly. The core problem is that arbitrary ids are assigned to nodes, which have to be looked up. This is done by the expensive joins.

However, utilizing a *hash function*, IRI strings can be mapped to distinct ids. Given such a function, it will always return the same id for the same string. This eliminates the need to keep a mapping from node name to its id and looking it up, making data loading almost trivial, as seen in Listing 4.2.

Java supplies a `.hashCode()` function which returns a 32-bit `int`.[1] Note however, that non-perfect hash functions do not guarantee mapping to unique ids. With the chosen Java hash function for strings, collisions can occur as well. While unlikely up to certain input sizes - more specifically in this case: number of unique nodes in the dataset - increasing amounts of data also increase the probability for collisions. Therefore, the use of perfect, or at least safer hash functions might be considered.

[1] *http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html#hashCode()*

Listing 4.1: Generic RDF-Graph Loading Implementation

```
1  val triples: RDD[(String, String, String)] = ...
2
3  val nodes: RDD[(String, Long)] = triples
4      .flatMap( t => Seq(t._1, t._3) )
5      .distinct()
6      .zipWithUniqueId()
7
8  val subjectKeyedEdges: RDD[(Long, String, String)] = triples
9      .groupBy( _._1 )
10     .leftOuterJoin(nodes)
11     .flatMap{
12         case (_, (iter, uid)) => {
13             iter.map( t => (uid.getOrElse(0L), t._2, t._3) )
14         }
15     }
16
17 val indexEdges: RDD[(Long, Long, String)] = subjectKeyedEdges
18     .groupBy( _._3 )
19     .leftOuterJoin(nodes)
20     .flatMap{
21         case (_, (iter, uid)) => {
22             iter.map( t => (t._1, uid.getOrElse(0L), t._2) )
23         }
24     }
```

4.2.2 Transitive Rules

As it turns out, *transitive* inference rules can be applied in a graph-parallel manner by employing the *Pregel* model. Consider the pattern of transitive rules:

$$a\ p\ b \quad \& \quad b\ p\ c \quad \Rightarrow \quad a\ p\ c$$

where p is either *rdfs:subClassOf* or *rdfs:subPropertyOf*. Actually, when OWL is also considered, p is not restricted to these two concrete RDFS properties but can be any *transitive* property as defined in OWL [Ter Horst, 2005]. That is, any p in an RDF dataset which also contains the triple:

$$p\ \textit{rdf:type}\ \textit{owl:TransitiveProperty}$$

The implementation is fairly straightforward to understand. Take any node in the graph which has one or more *incoming* edges with the attribute *rdfs:subClassOf*. This node can send its name along those edges (in reverse direction) to its neighboring nodes. Let this be the first superstep.

Listing 4.2: RDF-Graph Loading Implementation using Java hashCode

```
1  val triples: RDD[(String, String, String)] = ...
2
3  val nodes: RDD[(Long, String)] = triples
4      .flatMap(t => Seq(t._1, t._3))
5      .distinct()
6      .map( node => (node.hashCode().toLong, node) )
7
8  val edges: RDD[(Long, Long, String)] = triples
9      .map( t =>
10         (t._1.hashCode().toLong, t._3.hashCode().toLong, t._2) )
```

In the next step, nodes may receive zero, one or more messages, containing the names of its superclasses. Removing the duplicates, a concrete node N stores those names as the set of its superclasses. N is a subclass of all nodes in that set. Furthermore, any node that is a subclass of N, is also a subclass of all superclasses of N.

Thus, node N sends its superclasses along its *incoming rdfs:subClassOf* edges, distributing subclass information across the graph. To prevent ever-growing messages and ensuring termination of the program, the nodes only send along newly received superclasses by calculating the difference of its existing set and the newly arrived nodes. If it receives no new superclasses, no message is sent. This way, only new information is passed along. See Listing 4.3 for an implementation with Spark.

This example specifically used the subClassOf-property, but it applies to all transitive properties. However, it is also restricted to one transitive property at a time. Furthermore, by taking only the subgraph of the RDF graph, which is connected by a single transitive property, for instance subClassOf, the edge attribute need not be checked, because every edge has the same attribute. Notice that this is basically the task of calculating the transitive closure of the graph, which, as mentioned before (see 3.3), is equivalent to the application of transitive rules. It is advisable to separately look at the three functions provided to the GraphX `pregel` method. They are the *vertex program* (lines 5 to 21), the function for *sending* messages (lines 23 to 29) and the function for *merging* received messages (line 31).

4.2.3 Adapted Lookup Approach

The remaining non-trivial RDFS rules can also utilize the graph structure but they do not lend themselves to fully-fledged Pregel-like solutions. Instead they can be implemented with neighbor-collecting and schema lookup. This is best explained schematically for the case of rule 2:

1. Every node collects all neighbors connected via an *outgoing* rdfs:domain edge.

2. Every node collects *all* its *outgoing edge attributes* (connected nodes are not required).

3. For every node n:

   a) For each edge attribute e collected, look up the node with that name e and get the nodes it collected in step *1*.

   b) Remove duplicates

   c) For every node m collected this way it can be inferred: n *rdf:type* m

This works exactly analogous for rule 3, replacing rdfs:domain with rdfs:range in step *1* and collecting all *incoming* edges in step *2*. Similarly, rules 7 and 9 can follow the same approach with slight adjustments. For instance, they need to also look up connected nodes in step *2*, not only the edge attribute.

Step *3a* also assumes, that arbitrary nodes can be directly looked up by their name. This is necessary because there exists no link between the edge attribute e and the node e (see 4.2). Unfortunately, GraphX does not provide a way to do this and a workaround has to be found.

Similarly to the MapReduce approach, the relevant schema triples (for example *domain*) can be collected in a local lookup map and broadcast across the cluster. This map can then be used for lookup in step *3a*. Additionally, step *1* would not be required anymore. Alternatively, to avoid the lookup map and achieve a solution purely based on a graph representation, the schema information can also be stored as additional edge attributes.

4.2.4 Drawbacks of Graphs and GraphX

One core advantage of MapReduce approaches is that especially the *N-Triples* format is well suited as input format. In Spark, the individual lines of the input file directly map to triple entries in an RDD. This collection can be filtered as well as easily joined with new triples: The result of a rule application can readily be appended to the existing triple set with no or only minor processing required.

This is not as straightforward when working with GraphX. As discussed before, the input first has to be transformed to a graph format. The point of rule application is to infer new triples, these translate to new edges in the graph. However, GraphX offers no way to dynamically add new edges to the graph during processing. Hence, the data inferred is gathered on the vertices. After a rule has been applied, this data needs to be collected from the vertices and inserted to the graph as edges, which obviously incurs additional performance overhead. This has to be done after every rule.

Additionally, GraphX offers no efficient way of querying nodes, and no option at all during processing. The workaround to this problem has been discussed in the previous subsection 4.2.3.

Implementation:	Original Cichild	Node-local Cichild	Graph Implementation
Avg. Duration:	5s	13s	15s
%	1	2.6	3

Table 4.1: Domain and Range Inference, rdfs2 & rdfs3

4.3 Test Results

Various tests were conducted, unfortunately however not on a distributed cluster. As such, these results are of very limited significance in terms of actual performance in a distributed environment. As such, these values are analyzed as comparative indicator. The input RDF data for the tests in tables 4.1 and 4.2 is a generated dataset with approximately 13 million triples[2]. Testing of transitive closure was conducted differently and will be explained in the respective paragraph.

4.3.1 RDFS Rules 2 and 3 in Comparison

Table 4.1 shows the average duration for the application of RDFS rules 2 and 3 (*domain* and *range*) with the different implementations discussed in the previous sections. It can be seen that the original implementation in Cichild performs significantly better than both approaches suggested in this paper. The node-local approach, discussed in 3.4.2 is expected to be slower but as a trade-off it significantly reduces the number of duplicates.

As duplicate removal is performed locally, no further joins have to be performed for this part of the rule application. On the other hand, the input data has to be preprocessed (removal of duplicates, grouping). The test results thus suggest that preprocessing of the data is potentially costlier than simply removing duplicates at the end.

The graph approach also handles duplicates locally. Output is collected on vertices; a vertex v contains all derived tripels with the pattern $v\ p\ o$, where v is constant and p and o vary. Thus, similarly to the node-local approach above, it is guaranteed that all triples that begin with v as a subject are on the vertex, which then can perform removal of duplicates individually.

4.3.2 RDF Data Loading and Graph Building

Table 4.2 illustrates the significant performance gain when loading a graph by hashing the IRI to an identifier instead of assigning arbitrary unique identifiers and self-joining the dataset multiple times. The input dataset contains approximately 315,000 unique nodes that are hashed to an identifier. For the given input, no hash collision did occur. The poor performance of the initial approach would have posed a major obstacle to a full, graph-based solution in GraphX. Because GraphX does not offer a direct way to add edges and nodes to an existing graph, a new one would have had to be built after every rule application which would have been very costly and slow.

[2]Generated with *http://swat.cse.lehigh.edu/projects/lubm/*

Implementation:	Self-Join	Hashing
Avg. Duration:	22s	3s
%	7.33	1

Table 4.2: Graph Loading Joining vs. Hashing

Input Nodes	Input Edges/Triples	Dur. MapReduce	Dur. Graph
100	180	3s	2s
200	1,100	26s	2s
400	1,800	DNF	3s
1,000	3,800	DNF	3s
10,000	43,000	DNF	13s
20,000	83,000	DNF	24s

Table 4.3: Transitive Closure

4.3.3 Transitive Closure Comparison

The transitive closure tests, the results of which can be seen in Table 4.3 were conduced with randomly generated graphs, as provided by the GraphX API. The table of results shows concrete test results with a given input graph. The number of edges directly translates to the number of triples, which served as input for the MapReduce approach.

Interestingly, the graph approach performs significantly better for the input values used. In fact, the execution time of the original Spark implementation,, utilizing smart transitive closure, quickly increased, and starting from approximately 1,500 input triples, the program did not finish within five minutes. The graph-based approach on the other hand, could handle up to 20,000 vertices and 83,000 edges on a *single* working node.

Again, as testing was limited to a single node. Table 4.3 could however hint at a possibly better performance of a graph based closure approach towards transitive rules, compared to the MapReduce implementation.

Listing 4.3: Transitive Closure on a Graph

```
1  val graph: Graph[(String, Set[String], String), String]
2  val closure = graph
3    .pregel("INIT", Int.MaxValue, EdgeDirection.In)(
4    // VERTEX PROGRAM
5    (id, currValue, newMesg) => (currValue._1,
6      // calculate new Set of reachable nodes
7      if (newMesg == "INIT")
8        Set[String](currValue._1)
9      else if (newMesg.contains(",")) {
10       val split = newMesg.split(",").toSet
11       currValue._2 ++ split
12     } else currValue._2 + newMesg,
13     // calculate difference newNodes - existingNodes
14     if (newMesg.equals("INIT")) {
15       currValue._1
16     } else if (newMesg.contains(",")) {
17       (newMesg.split(",").toSet[String] -- currValue._2)
18         .mkString(",")
19     } else {
20       if (currValue._2.contains(newMesg)) "" else newMesg
21     }),
22   // SEND MESSAGE
23   triplet => {
24     if (triplet.dstAttr._3 == "") {
25       Iterator.empty
26     } else {
27       Iterator((triplet.srcId, triplet.dstAttr._3))
28     }
29   },
30   // MERGE MESSAGE
31   (a, b) => (a + "," + b))
```

5

Limitations

The scope of this paper is limited to RDFS entailment rules. A practical solution should however also be able to handle the *Ter Horst-Ruleset* [Ter Horst, 2005]. Furthermore, testing of concrete graph implementations was mostly focused on functionality and feasibility. For more meaningful results in terms of performance, comparing the graph implementations with the original MapReduce approaches, further testing should be conducted.

In principle, GraphX provides all the operations required to implement the graph solutions. However, it mainly focuses on processing graphs as a whole or subgraphs. As such, it is well suited for aggregating values across a graph and implementing graph parallel algorithms. On the other hand, it is not optimized for directly accessing individual nodes or edges. Furthermore, building on Spark RDDs, GraphX graphs are immutable, hence there is no efficient way to add individual nodes or edges.

6

Future Work

Some potential next steps are evident from the limitations given in 5. Towards a more comprehensive graph-based solution, OWL and the Ter Horst rule set should also be considered. They introduce further challenges, such as rules with three antecedent triples.

Furthermore, an efficient approach to collecting data from the nodes after applying a rule, and then merging it into the existing data is another important aspect to go from individual rule application to a complete inference engine.

Finally, further tests in a distributed cluster and with larger RDF datasets could yield data that is more conclusive, in relation to real-world applications.

7

Conclusions

It could be shown, that a graph-based approach towards distributed RDF reasoning is, in principle, possible. This is supported by several concrete implementations based on GraphX. Furthermore, limited test data indicates that these implementations run within reasonable time. In the case of the transitive RDFS rules, the graph-based solution to computing the transitive closure performed significantly better than the existing implementation of Cichlid.

Analyzing the Cichlid implementation, a logical error was discovered and fixed. A potential issue regarding high amounts of duplicates and their elimination was also identified and an alternative was suggested. During tests, the alternative implementation of duplicate-handling performed slightly worse, but it reduces the number of duplicates and allows them to be eliminated locally.

What remains unclear is how a graph-based implementation performs under real-world conditions, on distributed clusters with very large datasets and how it compares to existing approaches. Furthermore, it remains to be seen, how well the programs for individual rules integrate into a reasoning engine that iteratively applies several rules.

References

[Angles and Gutierrez, 2005] Angles, R. and Gutierrez, C. (2005). Querying RDF data from a graph database perspective. *Semantic Web: Research and Applications, Proceedings*, 3532(c):346–360.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. *Sixth Symp. Oper. Syst. Des. Implement.*, 51(1):107–113.

[Gonzalez et al., 2014] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX : Graph Processing in a Distributed Dataflow Framework. *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 599–613.

[Gu et al., 2015] Gu, R., Wang, S., Wang, F., Yuan, C., and Huang, Y. (2015). Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark. *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 700–709.

[Leskovec et al., 2014] Leskovec, J., Rajaraman, A., and Ullman, J. D. (2014). *Mining of Massive Datasets*. Cambridge University Press, 2 edition.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, pages 135–146.

[Ter Horst, 2005] Ter Horst, H. J. (2005). Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics*, 3(2-3):79–115.

[Urbani et al., 2012] Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., and Bal, H. (2012). WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Journal of Web Semantics*, 10:59–75.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10.

List of Figures

List of Tables