University of Zurich^UZH

Thesis May 8, 2013

Extending Rdfbox with Named Graphs Support

Alessandro Rigamonti
of Zurich ZH, Switzerland

Student-ID: 08-922-601
alessandro.rigamonti@gmx.net

Advisor: **Cosmin Basca**

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

Acknowledgements

First of all I would like to thank Professor Abraham Bernstein, Ph.D., for giving me the chance to pursue my bachelor thesis at the Dynamic and Distributed Information Systems Group (DDIS).

I would also like to express my gratitude to my supervisor Cosmin Basca for giving me useful advice throughout the thesis. Furthermore I would like to thank my friends at the Department of Informatics (IFI) for helping me out when I found myself stuck.

Last but not least, I am deeply grateful to my family. During my years of study they never put me under pressure and supported me whenever needed.

Zusammenfassung

Das Semantische Web (engl. Semantic Web) ist ein Konzept zur Erweiterung des World Wide Webs mit semantischen Informationen, so dass diese von Computern "verstanden" werden können. Eine technische Umsetzung hierfür bietet das Resource Description Framework (RDF). Dabei werden die Informationen in Form von Tripeln (Subjekt, Prädikat, Objekt) abgespeichert. Solche Daten können mit der Datenbanksprache SPARQL Protocol And RDF Query Language (SPARQL) abgefragt werden. Es existieren zahlreiche native, relationale Indexstrukturen für RDF. Ein leseoptimiertes Modell stellt Hexastore dar, wobei für die sechs Tripelpermutationen je ein Index erstellt wird. Rdfbox ist ein Tool für die Abfrage und das Speichern von RDF-Daten. Es setzt das Konzept von Hexastore um und verwendet dabei auf der Ebene des physikalischen Speichers B+ Bäume. SPARQL 1.1 brachte neu das GRAPH Schlüsselwort, das Abfragen nach der Quelle der RDF-Daten erlaubt, mit sich. Ziel dieser Arbeit ist es, Rdfbox um die Unterstützung dieses GRAPH Schlüsselworts zu erweitern.

Abstract

The Semantic Web is a concept for extending the World Wide Web with semantic information in a way that computers are able to "understand" it. A technical realization for that is provided by the Resource Description Framework (RDF). Thereby the information is stored uniquely in the form of triples (subject, predicate, object). Such data can be queried using the SPARQL Protocol And RDF Query Language (SPARQL). There are a number of native and relational-based indexing schemes for RDF. A read-optimized schema is depicted by Hexastore, whereat for all six triple permutations a separate index is created. Rdfbox is a tool for querying and storing RDF data. It extends the Hexastore indexing paradigm with update/write support by relying on B+Trees as the main physical index storage. SPARQL 1.1 brought the GRAPH clause that allows for querying the source of RDF data. The goal of this thesis is to extend Rdfbox with support of that GRAPH clause.

Contents

1 Introduction

Websites are designed in a way that they are understandable for humans. There are text sections, graphics, tables and other instruments in order to convey contents to the visitor. If a website displays "opening hours" with a table underneath containing data like "08:00 - 19:00", most of us will probably conclude that the shop has opened from 08:00 to 19:00. If instead of "opening hours" there was written "opening period" or "opening time", we would still come to the same result. Neither matters the positioning. If the title was on the left of the table we would still know it relates to the table. With our cognitive capabilities we are able to interpret such information.

Machines do not have these cognitive capabilities. They can read such information but they do not understand it. "opening hours" is just a sequence of letters and does not describe at what time the shop opens and closes. This is where Semantic Web [4] comes into play. It does not make machines more intelligent but it provides machines with the meaning of data. So when a computer finds the letters "Zurich" on a shop's website it will know whether they describe the location of the shop or the home town of an employee or something totally different. That brings an enormous advantage: machines can perform tasks that would normally be done exhaustively by humans. They can for example look for shops in a specific town with specific opening hours that sell e specific product to a specific price.

In order to make data understandable for machines we need a standard which is in our case the Resource Description Framework (RDF) [7]. Thereby information is stored in triple form, where every triple consists of a subject, a predicate and an object. This model is sufficient to express a lot of meta information and is at the same time very simple. For the above example one triple could look like: shopXY location Zurich. This tells us that the location of shopXY is Zurich. A tool for managing such RDF data is Rdfbox, which was developed at the University of Zurich by the Distributed Information Systems Group (DDIS).

RDF data can be represented as a graph where subjects and objects are nodes and predicates are edges. Since it can be useful to not only have information about single elements but also about whole graphs, Semantic Web extends the RDF model with named graphs. Therewith we can for example assign triples to a graph, whose name stands for the source of the triples. So far Rdfbox did not support named graphs. The goal of this thesis is to extend Rdfbox with that functionality.

Outline The structure of this thesis is as follows. Chapter 2 introduces the basic concepts of semantic data management and provides an introduction to Rdfbox, focussing on the index structure and the query processing mechanism. Chapter 3 explains the principle of named graphs in detail since this is the main topic of the thesis. In chapter 4 the technical considerations and the implementation are described. The main focus in that chapter concentrates on the index structure and the query processing. After that, in chapter 5 it is shown how the implementation is tested to make sure it fulfils the task. The thesis is concluded with chapter 6 which is about future work to be done with Rdfbox.

2

Background

This chapter discusses first some general aspects of the Semantic Web including the RDF standard and the SPARQL query language. We than introduce and detail Rdfbox. This provides the necessary knowledge to understand the main part of the theses, meaning the implementation of the necessary functionality to cope with named graphs.

2.1 Semantic Data Management

The Resource Description Framework (RDF) is a standard of the World Wide Web Consortium (W3C) originally designed for storing meta data [2]. Since it allows for flexible modelling of information it is a central component of the Semantic Web. Section 2.1.1 explains the concept of RDF and section 2.1.2 introduces the query language used for RDF. Eventually it is shown why there is a need for special triple stores.

2.1.1 Data Representation

RDF is represented in the form of triples, which consist of a subject, a predicate and an object. Considering triple data subjects, predicates and objects are referred to as elements. Let us consider an example: Gwendoline likes Farin. In this case, Gwendoline is the subject, likes is the predicate and Farin is the object. This triple gives us some information about Gwendoline, namely that she likes Farin. Except the reverse statement that Farin is liked by Gwendoline there's no information about Farin, because we do not know whether he likes Gwendoline or not.

An object in one triple can be a subject in another and vice versa. Farin, the object in Gwendoline likes Farin, is the subject in Farin is Bored. This leads to the awareness that RDF data can be considered a directed graph, in which subjects and objects are nodes and predicates are edges (see figure 2.1).

In practice triples do not look like the ones discussed before. There are certain formats the elements can take. A subject is always a resource, which in turn is a either a blank node or an uniform resource identifier (URI). In this thesis we just have to deal with URLs, such as <http://xmlns.com/foaf/0.1/person1>. Objects can be resources as well as unicode string literals such as "Alice". Finally, predicates are always URIs.

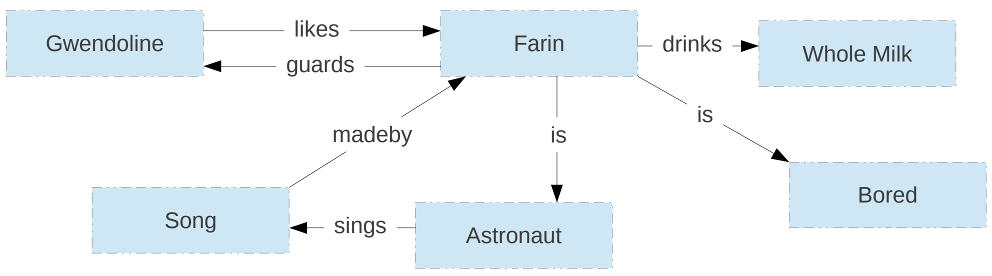

Figure 2.1: RDF triples illustrated in a directed graph.

Now we know how single triples look like in practice. But usually we want to provide RDF data that consists of many triples. There has to be a standard way to do that in order to make RDF tools able to scan that data. There are several conventions for that, for example the XML, the TURTLE or the N-Triples format [2]. Listing 2.1 shows some triples in TURTLE format. Thereby each triple is placed at a separate line and each line ends with a dot. Further it is allowed to define some abbreviations using the keyword @prefix. foaf:person1 in listing 2.1 for example is equivalent to <http://xmlns.com/foaf/0.1/person1>. The data in that listing applies FOAF, which is a scheme that uses RDF to describe social networks [5]. The N-Triples format, which is relevant for this thesis, uses the same structure as the TURTLE format except that the definition of prefixes is not allowed.

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

foaf:person1 foaf:knows foaf:person2 .
foaf:person1 foaf:name "Alice" .
foaf:person2 foaf:name "Bob" .
```

Listing 2.1: RDF triples in a TURTLE file.

2.1.2 SPARQL Query Language

This section is about querying data from existing RDF databases. Let us assume there is a database that contains the triples from listing 2.1. We now want to get some information out of it, for example all the people that are known by person1. For that purpose we need a query language, which in the case or RDF is the SPARQL Protocol And RDF Query Language (SPARQL) [9].

Since RDF data is a graph, SPARQL was designed to find graph patterns in a RDF database. Graph patterns are sets of triple patterns. A triple pattern in turn consists of a subject, a predicate and an object. Considering triple patterns subjects, predicates and

```
SELECT ?person
WHERE {
        <http://xmlns.com/person1> <http://xmlns.com/knows> ?person .
}
```

Listing 2.2: SPARQL query that returns the person known by person1. The URIs are shortened in order to keep the triple pattern at one line.

objects are referred to as variables, which all can be either bound or unbound. Bound means we give a specific value to a variable in the triple pattern that the element in the triple data has to match. Unbound means the element can be anything. An example of such a graph pattern is shown in the WHERE clause in the SPARQL query in listing 2.2 (the URIs in that query are shortened in order to keep the triple pattern at one line). This one consists of only one triple pattern of which the subject and the predicate variables are bound. Finding such a graph pattern in a RDF dataset means to find a set of triples in the dataset that matches that graph pattern. There can be several sets that match the same pattern. In our example such a set consists of only one triple because the graph pattern contains just one triple pattern. The first triple in listing 2.1 matches because the subject and the predicate values are the same as in the pattern. Moreover it is the only matching triple in the database.

The graph pattern is always put into the WHERE clause. In the SELECT clause there is specified which unbound variables of our graph pattern have to be returned. In our example that variable is ?person which is the object variable in our triple pattern. The values are then taken out of the result sets. So the object value of our result set (the first triple in listing 2.1) is returned which is <http://xmlns.com/foaf/0.1/person2>.

In order to make triple patterns clearly represented SPARQL provides a PREFIX clause that allows for defining abbreviations. These look similar to the prefixes in TURTLE files as the query in listing 2.3 shows. In this query there are two triple patterns. Thus, our result sets consist of two triples. Moreover, object value of the first triple and subject value of the second one have to be the same. That is because the unbound variable ?person is the object variable of the first and at the same time the subject variable of the second triple pattern. This is called a join on the variable ?person. So the query returns all the names (?name) of the people (?person) that are known by foaf:person1. Again there is only one result set, namely the one consisting of the first and the third triple in listing 2.1. The returned value is "Bob".

There are various other keywords and more complex graph patterns, but the basic principle of SPARQL shown by these examples should provide enough understanding for the moment. The GRAPH keyword is discussed separately and in more detail in chapter 3 since it is an important part of the thesis.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        foaf:person1 foaf:knows ?person.
        ?person foaf:name ?name.
}
```

Listing 2.3: SPARQL query that returns the name of the person known by person1. The PREFIX keyword is used to make the triple patterns clearly represented.

2.1.3 Triple Stores

The structure of a database can be a critical issue when the amount of data becomes bigger and bigger. Some queries require a lot of operations, so resolving these queries for big databases can be very expensive. Thus, during many years relational databases were optimized and now come up with good performances.

RDF data can be stored in such relational databases. The easiest approach is to store everything in one table with the three attributes subject, predicate and object. Unfortunately that table becomes very long for big datasets, which leads to bad performance when doing self joins. Since SPARQL queries require a lot of self joins in case there is just a single table, this method is not satisfying. Another approach for storing RDF data in relational databases is Vertical Partitioning. Thereby for every predicate a separate table with the two attributes subject and object is created. For the above query examples it would work well, but as soon as the predicate is unbound and joins have to be performed on it, that approach causes problems.

So a relational database cannot be a satisfying solution for big RDF datasets. That is why special triple stores [10] are needed. One approach of a triple store is Hexastore that is explained in section 4.1.1.

2.2 Rdfbox - A Semantic Knowledge Repository

Rdfbox is a database management system for RDF data programmed in Python and C. This section discusses first the index structure of Rdfbox that is based on Hexastore. Then the query processing mechanism is explained and eventually an overview of the external components is given.

2.2.1 Hexastore

Hexastore [11] is a triple store created at the University of Zurich and implemented in Rdfbox. The philosophy of Hexastore is to accept a bigger demand of space in order to make resolving queries faster. So for any permutation there is a separate index. Since there are three elements, this results in the six permutations spo, sop, pso, pos, osp and ops, where spo stands for subject/predicate/object, sop for subject/object/predicate and

s	p	o
s1	p1	o1
s1	p1	o2
s1	p2	o1
s1	p2	o3
s2	p2	o1
s2	p4	o4

Table 2.1: Example triples using short names instead of URIs or literals.

s	count
s1	2
s2	2

(a) spo0 index.

s	p	count
s1	p1	2
s1	p2	2
s2	p2	1
s2	p4	1

(b) spo1 index.

Table 2.2: Graphical representation of spo0 and spo1 index according to table 2.1.

son on. Every index in turn consist of three levels, each in a separate index file. So we have 18 index files. Let us consider the spo index as an example. Table 2.1 shows a bunch of triples sorted lexicographically in spo order. Level 0 of the spo index is called spo0 and just contains the distinct subject values (see table 2.2). The count of a certain subject in the spo index says how many different predicates there are for that subject. In this case there are two predicates (p1 and p2) for the subject s1 and two predicates (p2 and p4) for the subject s2. The spo1 index contains the distinct subject predicate combinations. The first combination, s1p1, has two objects (o1 and o2) and therefore count 2. The spo2 index just looks like 2.1 assumed no triple exists twice in that table. It contains the subject, predicate and object values sorted in that order. A count for the spo2 index does not make sense because there is no further element to count.

To demonstrate what these indexes are good for let us consider the query in listing 2.3. As discussed earlier we try to find two triples that match the graph pattern. Thereby we first look for single triples that match the first triple pattern. Then we do the same for the second triple pattern and after that we take only the triple combinations where the object value of the first one is the same as the subject value of the second one. So actually we just need all the object values for the given subject predicate combination foaf:person1 foaf:knows and all the subject values for the given predicate foaf:name in order to compare them afterwards. We can get the object values with the spo2 index and the subject values with the pso1 index. Since every index is sorted lexicographically, we get these values not only quickly, but also in a sorted manner. This allows for a very fast join that gives us all possible values for ?person. For every value we can now quickly get the possible values for ?name using the spo2 index. Since there is an index for every permutation there is fast access to the required values no matter how the triple pattern looks like. The count values are used for selectivity reasons. In the above discussed query there has to be decided whether to resolve ?person or ?name first. Section 2.2.2 explains how this decision is made based on the count values.

The actual URIs and literals are not stored directly in the index files. Instead they are dictionary encoded. So all elements first get an unique identifier (id) which is then written into the index files. Further, Hexastore and thus Rdfbox saves some space at index level 2. If in a triple pattern subject and predicate are bound and the object is unbound, the spo2 index can be used. Yet the pso2 index can be used as well. Therefore just one of these indexes is needed, as a shared index. The same thing counts for sop2 and osp2 and for pos2 and ops2 index files. Without shared indexes there are 18 index files (six permutations with three levels each). However, with shared indexes there are not 18, but 15 index files.

2.2.2 Query Processing

The core of Rdfbox are the Engine and the Index Manager. The actual query computing is done in these two modules. The Engine recursively resolves variable after variable. Thereto it needs the Index Manager which is the interface between Engine and database.

First of all the Engine gets a graph pattern which is a list of triple patterns. The task of the Engine is to give some valid values to the unbound variables or in other words resolve the unbound variables. They are resolved one after another, so first all the joins for the first variable are made as described in section 2.2.1, then all the joins for the second and so on. All variables resolved means there are a certain number (can be zero) of result tuples consisting of a value for every unbound variable.

The exact mechanism is shown in figure 2.2. The iterate function starts with the update of the cardinalities of the variables. Thereby for each variable the count of each triple pattern the variable is in is considered. For example the count value of ?p in <s1> ?p ?o according to table 2.1 is taken from spo0 index (see table 2.2) and is 2. Based on these cardinalities the most selective variable is chosen to be joined. The join itself is done by the Index Manager because the index files on the bottom have to be accessed. The Engine proceeds with the list of the possible values for that variable gotten from the Index Manager. For every value in the list is the following is done: it is temporarily written into that variable in all the triple patterns, the iterate function is called recursively and after the recursive call ended it is removed from the variable in order to try the next value in the list. So when the process starts over in the recursive call that variable is bound and the next one can be chosen to resolve. At that point it is obvious why there is a need for cardinalities. The iterate function is called for every single value in the list and variables with low counts produce less values out of the join.

The function is called recursively until either the the join returns no more values or all variables are resolved. In the former case the function ends and thus jumps one level up and tries the next value. In the latter case it emits the result and then jumps one level up and tries the next value.

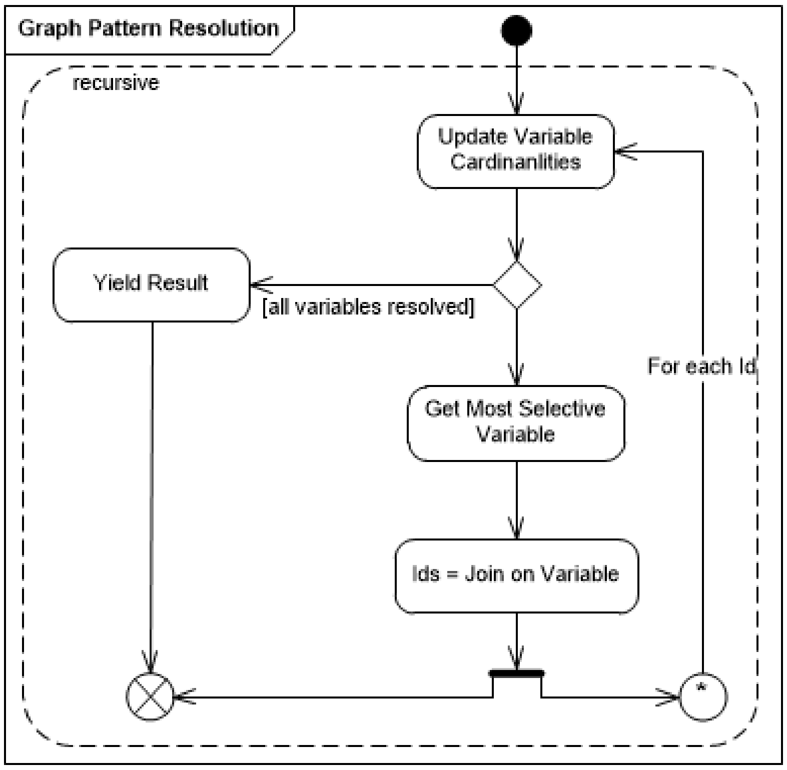

Figure 2.2: UML activity diagram of the Engines control flow.

2.2.3 External Libraries

There are many libraries used in Rdfbox. An overview over the most important ones follows here.

Cython

Cython [3] is a programming language that is a superset of Python. Cython code is written into Python files besides Python code (if Python code is necessary). The Cython code is translated once to C code then compiles with the common C compilers. The Python code runs on the usual Python interpreter and is able to call Cython functions. Thus, Cython is a way to combine Python and C code.

One reason why this is useful is the performance of efficient C code. It runs faster than Python code because it does not have to be interpreted at runtime any more. In performance critical parts of an application where there is a lot of runtime, C code enables a significant speed up. Another reason is the use of native C libraries. As Cython code is translated to C code it allows for including any library with a C API.

Tokyo Cabinet

Tokyo Cabinet [8] is such a library with a C API. It provides routines for managing physical databases. There are amongst others hash tables and B+ trees. The organisation of the indexes - namely which indexes are created, which index is used in what situation and so on - is done by Rdfbox itself following the concept of Hexastore. But the index files on the bottom, for example spo0, are files provided by Tokyo Cabinet organized in B+ trees.

Raptor

Another open source C library is Raptor. It provides a set of parsers that generate RDF triples by parsing different kinds of RDF data files. There are many supported syntaxes such as XML, N-Triples, TURTLE and so on.

Rasqal

Rasqal [1] is a C library as well. It is used for parsing SPARQL queries and works closely together with Raptor. It supports almost every SPARQL 1.1 query, including queries containing the GRAPH keyword which is discussed in chapter 3. It creates a rasqal query object from which any necessary information about the query can be received. That is for example the top graph pattern with all its sub graph patterns as well as lists of bound and unbound variables and so on.

3

Named Graphs

As we learned in section 2.1, RDF data consists of triples that have a subject, a predicate and an object. With this structure we can formulate descriptive statements like Gwendoline likes Farin. But what about the statement Rodrigo says Gwendoline likes Farin? Trying to represent that in RDF standard we have to divide this information up into several triples. The first one is obvious: Gwendoline likes Farin. But then the problem occurs. Rodrigo says describes not a single element but a whole triple. This chapter shows how RDF is extended to solve this problem and how SPARQL deals with it.

3.1 Semantics

First of all, is this kind of information even useful? Yes, consider thereto the following example. Let us assume the statement Gwendoline likes Farin" is true. Then the additional statement Gwendoline hates Farin does not make sense. These two statements are mutually exclusive. But the two statements Rodrigo says Gwendoline likes Farin and Bela says Gwendoline hates Farin are not. And that is a case that makes sense in practice, because depending on the *source* there can be different, even mutual exclusive, descriptions for the same subject.

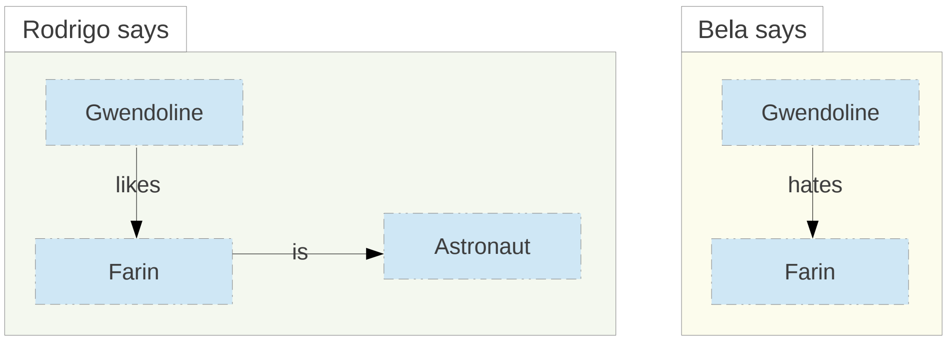

Figure 3.1: RDF triples in two different named graphs.

That is where named graphs come into play [6]. As shown in section 2.1 a set of RDF triples can be represented as a graph. The above problem is solved giving such a graph a name. One graph would then be named Rodrido says and the other one Bela says (see figure 3.1). From now on any triple can be part of a named graph.

There is an easy way to additionally provide named graph information. Instead of storing triples we store quads, where the fourth element is the *context* that says what named graph the triple belongs to. A format for that is the N-Quads format. It looks like the N-Triples format except that there is one more element per line (see listing 3.1). The @prefix keyword is actually not allowed but it helps to make the listing clearly arranged.

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix data:   <http://example.org/foaf/> .

foaf:person1 foaf:knows foaf:person2 data:alice .
foaf:person1 foaf:name "Alice" data:alice .
foaf:person2 foaf:name "Bob" data:bob .
foaf:person3 foaf:made data:bob data:charlie .
```

Listing 3.1: RDF quads in a TURTLE file with graph names data:alice and data:bob.

3.2 SPARQL

SPARQL 1.1 supports named graphs [9]. Thereto the GRAPH keyword is used as shown in listing 3.2. It contains a context variable and is wrapped around common graph patterns as we know them from section 2.1.2. The context variable specifies the named graph in which we are looking for the graph pattern within GRAPH clause. In other words the graph pattern within that GRAPH clause is matched only against quads having the same context value as the context variable of that GRAPH clause. Only the first two quads in listing 3.1 belong to the named graph data:alice. Thus, "Bob" is filtered out and just "Alice" is returned.

But the variable in the GRAPH clause does not necessarily have to be bound. It is possible to query the context of certain graph patterns as shown in listing 3.3. The result for that query is data:bob because it asks in what named graphs that graph pattern is findable.

More precisely the variables in GRAPH clauses act the exact same way as the other variables. Such a variable can be in a GRAPH clause and at the same time in a common triple pattern. The query in listing 3.4 first asks for all possible context values of a certain graph pattern. These values are then joined on the object values of the triple pattern ?person foaf:made ?context. Namely the query returns the person who made the named graph in which the triple pattern foaf:person2 foaf:name "Bob" is findable. According to listing 3.1 the value foaf:person3 is returned.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
PREFIX data:   <http://example.org/foaf/>
SELECT ?name
WHERE {
        GRAPH data:alice {
                ?person  foaf:name ?name.
        }
}
```

Listing 3.2: SPARQL query that returns all the names within the named graph data:alice.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?context
WHERE {
        GRAPH ?context {
                foaf:person2  foaf:name "Bob".
        }
}
```

Listing 3.3: SPARQL query that returns the context of a certain triple pattern.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE {
        ?person  foaf:made ?context.
        GRAPH ?context {
                foaf:person2  foaf:name "Bob".
        }
}
```

Listing 3.4: SPARQL query that returns the person that made the named graph that contains the triple foaf:person2 foaf:name "Bob".

In section 2.1.2 we defined a graph pattern as a set of triple patterns. From now on a graph pattern is a set of triple patterns and GRAPH clauses. There is no limitation in the number of GRAPH clauses or triple patterns and the order does not matter. The graph pattern within the SELECT clause is called top graph pattern. Within the GRAPH clause there is placed another graph pattern, which is what we call a sub graph pattern. For that sub graph pattern the same rules apply as for the top graph pattern. An example query with two GRAPH clauses is shown in figure 3.5. This query returns foaf:person1.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
PREFIX data:   <http://example.org/foaf/>
SELECT ?p1
WHERE {
        ?p1 foaf:name "Alice".
        GRAPH data:alice {
                ?p1 foaf:knows ?p2.
        }
        GRAPH data:bob {
                ?p2 foaf:name "Bob".
        }
}
```

Listing 3.5: SPARQL query with two GRAPH clauses.

3.3 Quad Pattern Representation

So far Rdfbox considered the execution of basic graph patterns (BGP) only. Thereby SPARQL queries were represented as a list of triple patterns that were matched against triple data. With the named graph support we need a new representation of SPARQL queries that considers context variables as well. Since we have now quad data the most intuitive way is to represent SPARQL queries as a list of quad patterns. The fourth variable is then the context variable of the named graph the triple pattern belongs to. If the triple pattern is not part of any named graph, the context variable is set to "None". This gives us a unique query representation because equivalent queries have the same quad pattern representation. The query in listing 3.5 for example has the quad pattern representation shown in listing 3.6.

```
?person1 <http://xmlns.com/name> "Alice" None
?person1 <http://xmlns.com/knows> ?person2 <http://example.org/alice>
?person2 <http://xmlns.com/name> "Bob" <http://example.org/bob>
```

Listing 3.6: Quad patterns of the query in listing 3.5. The URIs are shortened in order to keep the quad patterns at one line.

4

Technical Considerations and Implementation

The implementation of named graphs support affects several components of Rdfbox. Special attention is given to the index structure which is the basis for the following sections. The loading process of .nq files into that index structure as well as the parsing process of queries with GRAPH clauses are discussed in brief. Another main focus concentrates on the query processing mechanism at the end.

4.1 Index Structure

The index structure and the query processing mechanism are strongly depending on each other. That is why some thoughts about query processing flow into this section. First it is explained what what it means for quads to apply the concept of Hexastore. Then it is shown how the number of index files can be reduced. Finally the basic approach is discussed.

4.1.1 Applying the Hexastore Indexing Scheme

Since Hexastore is designed for triples the index structure in Rdfbox has to be extended. One of the first thoughts is to apply the idea of Hexastore for quads. This means instead of creating an index for any triple permutation (spo, spo, ...) creating an index for any quad permutation (spoc, spco, ...). The number of indexes is then the factorial of 4 which is 24. This way we would just slightly adopt the Engine because the resolving mechanism in general would be same. Namely, with all the index permutations we could still resolve any variable at any time. Unfortunately 24 indexes is just too much, because updating, inserting and deleting operations become very expensive. Even using six indexes as Hexastore does is above-average, even if there are some shared index files on level 2. Therefore applying Hexastore for quads is no solution and we need to make changes that affect both the index structure and the resolving mechanism.

s	?s	?p	?o	spo0
	?s	<p>	?o	pso1
	?s	?p	<o>	osp1
	?s	<p>	<o>	pos2/ops2
p	<s>	?p	?o	spo1
	<s>	?p	<o>	sop2/osp2
o	<s>	<p>	?o	spo2/pso2

Table 4.1: Index files needed when resolving variables in spo order.

4.1.2 Reducing Index Files

So we have to look at what the basic solution could be. Hexastore brings two advantages. One is that for any triple pattern there is fast access to the sorted elements looked for (see section 2.2.1). The other one is the use of the count values which helps to determine the order of variables to be resolved. Let us assume we dispense with the second advantage. This would reduce the number of necessary index files because then we could determine one order in which the variables are always resolved.

Table 4.1 shows the necessary index files if the fixed order is spo. There are four types of triple patterns when resolving the subject. The first one is that none of the variables are bound, the second one is that just the predicate is bound, the third one is that just the object is bound and the fourth one is that the predicate and the object are bound. Each type needs one specific index file to receive the ordered values. The fourth type needs either pos2 or ops2 which are obviously shared index files. Then there are just two more types of triple patterns when resolving the predicate because the subject is already resolved and therefore bound. In the first one the object is not bound, in the second one it is. Eventually there is just one type left when resolving the object since the subject and the predicate are both resolved. Thus, if the order is given only 7 out of the 15 index files of Hexastore are required.

4.1.3 Basic Approach

What does that mean for indexing quads? The above example showed that the number of index files can be reduced dispensing with selectivity. The goal is to apply this idea for context values while keeping the Hexastore idea for subject, predicate and object values. This is possible by first resolving the subject, predicate and object variables of all quad patterns and after that resolving all the context variables. Thus, the triple patterns are resolved the exact same way as before, including selectivity. Thereto the six indexes from Hexastore are unchanged. As the example in section 4.1.3 showed, we can use always the same index for a certain variable if we know that this variable is always the last one to be resolved. In the case of context variable it is the spoc index. The following three use cases show how the spoc index is utilised. From now on, let us call the subject, predicate and object variables "triple variables".

Bound Context

So far the Engine just considered unbound variables when matching triple patterns. This is possible due to the index structure. When receiving the values for ?s in a triple pattern ?s <p> <o> with the aid of pos2 the bound variables <p> and <o> are both already taken into consideration. But now we have to deal with quad patterns like ?s <p> <o> <c>. When we first resolve the triple pattern, we get the subjects of all quads with <p> as a predicate and <o> as an object. But these quads may not have <c> as a context. So the Engine has to consider bound context variables as well and filter out the quads with a wrong context values. This can be done with the spoc index, because the triple variables are already resolved when filtering. More precisely we just need the spoc3 index file since we just need to check whether there is a certain context value for some given subject, predicate and object values.

Unbound, fixed Context

The other case is that the context variable is unbound. But here we have to be careful. If the same variable appears somewhere else as a subject, predicate or object (see table 4.2), the Engine has to handle that context as if it were bound. That is because the triple variables are already resolved when resolving the context and therefore there is already a fixed value for that context variable. So the same filtering is applied as with bound context variables.

Unbound, not fixed Context

The last case are unbound context variables that do neither exist as a subject, nor as a predicate, nor as an object. For that case another join becomes necessary. Table 4.2 shows an example, where there is a need for a join on the variable ?d. As the previous subsection showed we never have to join a context variable with a subject, predicate or object variable. This case here just appears when several context variables have to be joined amongst themselves. This is now similar to the object in the example in section 4.1.3. For the object we needed spo1 for selectivity and spo2 for getting the object values. So for the context we need spoc2 for selectivity and spoc3 for the getting the context values. Selectivity can still be useful because there can exist queries with several unbound, not fixed context variables.

Therefore the basic approach requires 17 index files, namely the 15 index files from Hexastore plus spoc2 and spoc3.

Bound				**Unbound**							
				Fixed				**Not Fixed**			
?a	<b>	<c>	<d>	<a>	<b>	<c>	?d	<a>	<b>	<c>	?d
				?d	<e>	<f>	N	<e>	<f>	<g>	?d

Table 4.2: One example for each of the three use cases of context appearance. N stands for None.

4.2 Loading .nq Files

Figure 4.1 shows the control flow for the complete process of index creating and loading. At the beginning the index permutations are determined. Based on that the Tokyo Cabinet index files are created iteratively. Additionally to the previous ones now the two index files spoc2 and spoc3 are created.

Then the RDF data, which now consists of quads, has to be loaded into these index files. Thereto primarily a .nq file is parsed so that there is a list of string based quads. As explained in section 2.2.1 these values are mapped to ids. That is why all the parsed values, including the context values, are encoded right after the parsing process. These encoded quads now occur in spo order, but in a unsorted manner since this is how they are provided by .nq files. In order load these ids later into the spo index files they are sorted lexicographically in spo order and then written into the spo buffer. Afterwards for each permutation the ids are reordered, sorted and written into the specific buffer iteratively. Furthermore the whole process from parsing the .nq file until writing the ids into the buffers is done in parallel. Eventually the ids are loaded into the TC index files using the six previous buffers and the new spoc buffer.

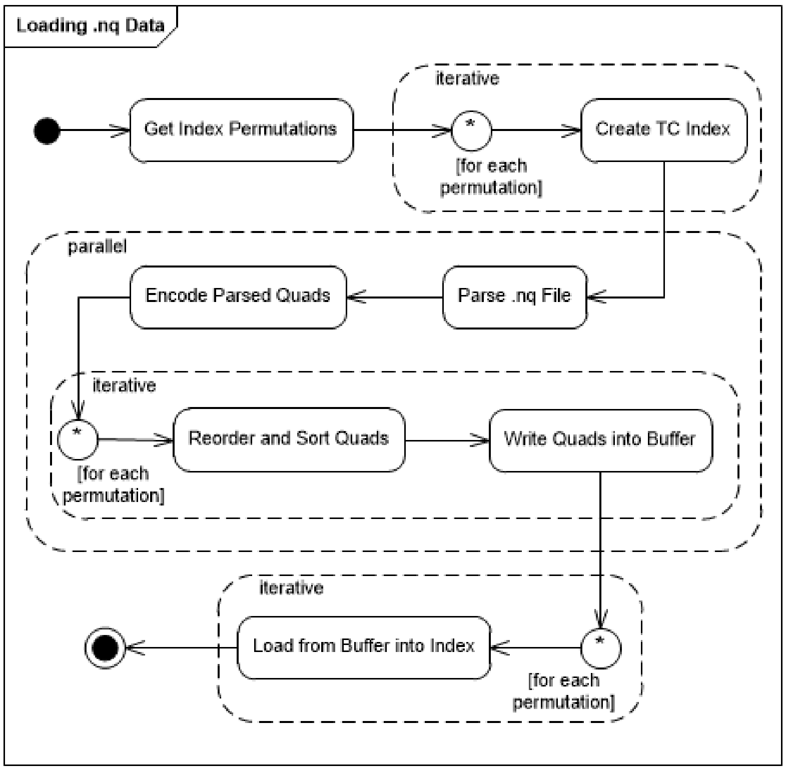

Figure 4.1: UML activity diagram of the loading process.

4.3 Parsing Queries

In Rdfbox the queries are parsed using the Rasqal library. As illustrated in figure 4.2 the query string is read and then the Rasqal objects are created. These objects provide lists of all triple patterns, bound and unbound variables and so on. However, we are interested in a list of quad patterns because these give us a unique representation of SPARQL queries as shown in section 3.3.

In Rasqal the representation of GRAPH clauses works as follows: the Rasqal query object contains a method that returns the top graph pattern as a Rasqal graph pattern object. These graph pattern objects take certain types, of which three are relevant for us. If the type is *basic* the graph pattern just contains a list of triple patterns, thus without any further sub graph patterns. If the type is *group* the graph pattern consists of several sub graph patterns. In our case these sub graph patterns can take the types *basic* and *graph*. If the type is *graph* the graph pattern represents a GRAPH clause. This graph pattern allows for calling the context variable as well as the one sub graph pattern hich in turn can take any of these three types.

So after the Rasqal objects are created the context variables are added to the triple patterns using a recursive function. At the beginning the top graph pattern is called. If type is *group* the function is called recursively for every sub graph pattern. If type is *graph* the context variable is temporarily stored and the function is called recursively for the one sub graph pattern. If type is *basic* and it is the top graph pattern the value "None" is added to the triple patterns in it. If type is *basic* and it is not the top graph pattern the temporarily stored context variable is added to the triple patterns.

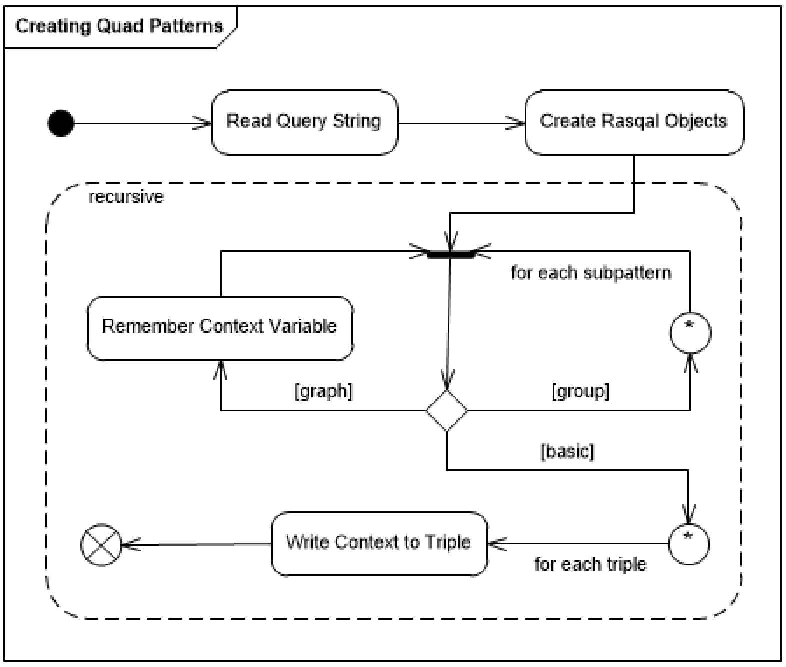

Figure 4.2: UML activity diagram of the creation of the quad patterns.

4.4 Query Processing

The index structure discussed in section 4.1 is designed for resolving the unbound triple variables first and then handle the bound and unbound context variables. That is why we have to treat variables differently.

The list of unbound triple variables remains the same. But additionally two lists are created. One is a list of unbound context variables described in the third use case, namely the ones that do not appear as a triple variable. The other list contains quad patterns. More precisely it contains the quad patterns with a context variable that is either unbound and at the same time appears as a triple variable or is just bound. Let us call these the fixed context quad patterns. These quad patterns have to be checked as soon as variables of the list of unbound triple variables and variables of the list of unbound context variables have all been resolved.

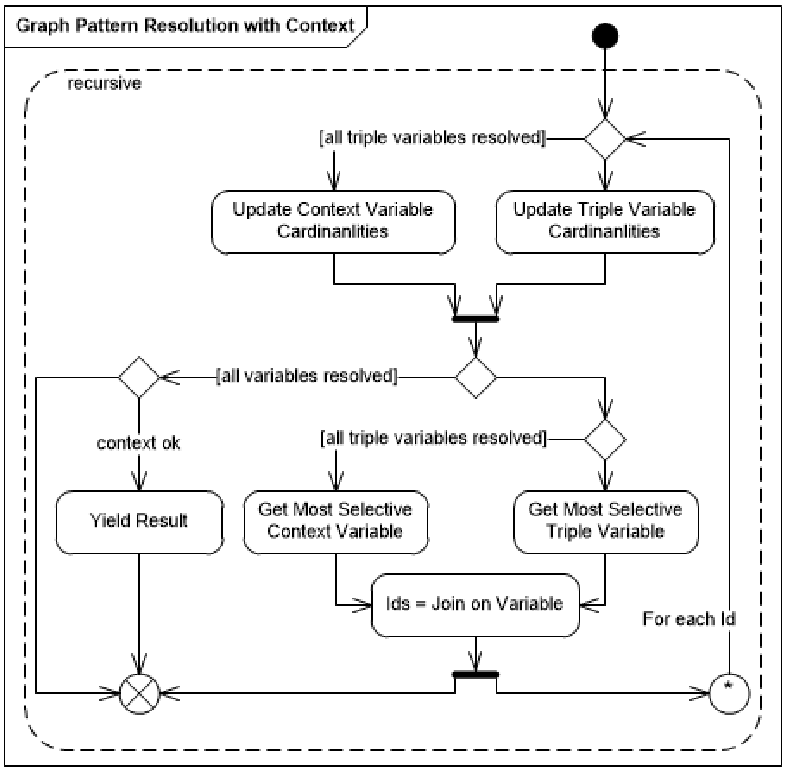
Graph Pattern Resolution with Context
recursive
[all triple variables resolved]
Update Context Variable Cardinanlities
Update Triple Variable Cardinanlities
[all variables resolved]
context ok
[all triple variables resolved]
Yield Result
Get Most Selective Context Variable
Get Most Selective Triple Variable
Ids = Join on Variable
For each Id
*

Figure 4.3: UML activity diagram of the Engines control flow including context variables.

Figure 4.3 illustrates the new process in the Engine. The update of the cardinalities is still the first thing to be done. However, that function has to be extended. If not all unbound triple variables are yet resolved, their cardinalities are updated. Otherwise the cardinalities of the unbound context variables are updated. The function that checks whether all variables are resolved now has to additionally check the list of context variables. Then the most selective variable is chosen. Again there has to be checked whether all unbound triple variables are resolved. If not, the most selective triple variable is chosen. Otherwise the most selective context variable is taken. Then the join on the chosen variable is performed and for every value gotten out of that join the iterate function is called recursively.

This way we first resolve all the triple variable just like before and then resolve the unbound, not fixed context variables. After that the fixed context quad patterns can be checked. This is done right before the result tuple is emitted. At that point of time all the variables in the quad patterns are bound (remember that the unbound, not free context variables were resolved as triple variables). Hence the quad patterns can be considered quads since they have an actual value for every variable. In order that a result tuple is valid, every quad in the fixed context quad pattern list has to exist in the quad database. If that is the case, we know that these context values exist with the triple values we calculated and we can emit the result. If that is not the case, the function ends and thus jumps one level up and tries the next value gotten out of the join.

5 Empirical Evaluation

The goal of the thesis was to extend Rdfbox with named graph support. In order to evaluate the result we have to conduct some functional tests. The difficulty here is the infinite number of queries that are syntactically correct. Therefore we try to categorize queries and then run a bunch of representative queries against some sample data.

```
PREFIX data: <http://localhost:8890/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?s
WHERE
  {
  GRAPH data:ns#graph2
     {
     ?s foaf:knows ?o.
     ?s foaf:name ?m.
     ?o foaf:name "Chris Bizer".
     }
  }
```

Listing 5.1: Test Query 1. Basic example query with bound context variable.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?s
WHERE
  {
  ?s foaf:knows ?o.
  ?s foaf:name ?m.
  GRAPH ?s
     {
     ?o foaf:name "Chris Bizer".
     }
  }
```

Listing 5.2: Test Query 2. Basic example query with unbound and fixed context variable.

The sample data does not have to be huge since it is a functional test. For that purpose we take a FOAF dataset with just below 500 triples. With a Python script some random context values are added to the triples in order to have some quad data. A few context values have to be changed manually so that the more complicated queries make sense. This way we know for every test query the results that should be returned and can compare these with the actual results gotten when running the queries.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?g
WHERE
  {
  GRAPH ?g
    {
    ?s foaf:knows ?o.
    ?s foaf:name ?m.
    ?o foaf:name "Chris Bizer".
    }
  }
```

Listing 5.3: Test Query 3. Basic example query with unbound and not fixed context variable.

Listing A.1 contains all queries that were run against that dataset. Seven out of these queries are explained in this chapter in order to show the categorisation of the test queries. In section 4.1.3 the three use cases for context variables were discussed. These are at the same time the main categories in the evaluation. So the first queries that are tested are the basic ones of each category. Listing 5.1 shows a simple test query with one bound context variable. Listing 5.2 shows one with one unbound, fixed context variable and listing 5.3 illustrates one with one unbound, not fixed context variable.

After the first test set with basic queries we can go one step further and test queries with two or more GRAPH clauses. Again, there is an example query for each category. Listing 5.4 shows the one with two bound context variables, listing 5.5 the one with two unbound, fixed context variables and listing 5.6 the on with two unbound, not fixed context variables. To complete the evaluation some long, complicated queries are tested. These queries combine nested GRAPH clauses with bound and unbound context variables and so on. An example is given in listing 5.7.

The results of the evaluation are listed in table 5.1. It shows the expected results for every of these seven test queries and whether the actual results were the same or not. As we can see the test queries returned the right results. Also, every test query shown in listing A.1 returned the expected results.

```
PREFIX data: <http://localhost:8890/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?m
WHERE
  {
  ?s foaf:knows ?o.
  GRAPH data:ns#graph2
    {
    ?s foaf:name ?m.
    }
  GRAPH data:ns#graph3
    {
    ?o foaf:name "Chris Bizer".
    }
  }
```

Listing 5.4: Test Query 4. Example query with two bound context variables.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?p
WHERE
  {
  ?g ?p ?f
  GRAPH ?g
    {
    <http://www.bizer.de#chris> foaf:name "Chris Bizer".
    }
  GRAPH ?f
    {
    <http://berrueta.net#me> foaf:name "Diego Berrueta".
    }
  }
```

Listing 5.5: Test Query 5. Example query with two unbound and fixed context variables.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?g
WHERE
  {
  GRAPH ?g
    {
    ?s foaf:knows ?o.
    }
  GRAPH ?f
    {
    ?s foaf:name ?m.
    }
  ?o foaf:name "Chris Bizer".
  }
```

Listing 5.6: Test Query 6. Example query with two unbound and not fixed context variables.

```
PREFIX data: <http://localhost:8890/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?m
WHERE
  {
  ?s foaf:name ?m.
  GRAPH ?p
    {
    ?s foaf:knows ?o.
    ?o foaf:name "Libby Miller".
    GRAPH data:ns#graph1
      {
      ?g ?p ?f.
      }
    GRAPH ?g
      {
      <http://www.bizer.de#chris> foaf:name "Chris Bizer".
      }
    GRAPH ?f
      {
      <http://berrueta.net#me> foaf:name "Diego Berrueta".
      }
  }
```

Listing 5.7: Test Query 7. Example query with nested GRAPH clauses and bound and unbound context variables.

Test Query	Expected Result Set	Matched
1	<http://www.ivan-herman.net/foaf#me> <http://localhost:8890/ns#graph3>	yes
2	<http://localhost:8890/ns#graph3>	yes
3	<http://localhost:8890/ns#graph2>	yes
4	<http://www.ivan-herman.net/foaf#me> <http://localhost:8890/ns#graph3>	yes
5	<http://xmlns.com/foaf/0.1/graphfriend> <http://xmlns.com/foaf/0.1/graphbro>	yes
6	<http://localhost:8890/ns#graph2>	yes
7	”Ivan Herman”	yes

Table 5.1: Index files needed when resolving variables in spo order.

6

Future Work

In order to extend Rdfbox with named GRAPH support the basic approach was implemented. This approach focusses on keeping the number of index files as small as possible, so just the two index files spoc2 and spoc3 have been added. Moreover, the number can even be reduced to one additional index file while keeping the same functionality. It is clear from section 2.2.1 that the two index files spo0 and spoc0 contain the same data. This is true for spo1/spoc1 and for spo2/spoc2 as well, except that spo2 does not have the count values spoc2 has. So for getting subject, predicate and object values we do not need both the spo2 and the spoc2 index file. Since we need the count values of spoc2 we would then omit spo2. spo0, spo1, spoc2 and spoc3 is the same as spoc0, spoc1, spoc2 and spoc3 so, generally speaking, we could replace the spo index by the spoc index. Thus, the basic approach could be implemented with 16 index files instead of 17.

Chapter 5 shows that the functionality requirements are fulfilled. But so far performance was not taken into consideration. In our approach the context variables are always resolved at the end. It is obvious that there are queries where this can lead to a performance issue. Let us thereto look at the first query in listing 5.1 and assume our database is big but contains very few quads with data:ns#graph2 as their context value. In that case Rdfbox needs to match the graph pattern within the GRAPH clause against the whole data before checking the context variable. This can be very expensive. If instead we could match the graph pattern only against triples in the named graph data:ns#graph2 the query would be computed much quicker.

One approach could be to add a cs, a cp and co index, where for example the cs index consists of the index files cs0 (context values plus subject counts) and cs1 (subject values for a given context value). Let us again consider the first query in listing 5.1 and assume that ?o is the first variable chosen to be resolved. As we learned in section 2.2.2 we start with a join on that variable. We would just have to include the co index in that join. Namely, this index provides the object values that occur in quads with data:ns#graph2 as their context value (which are not many according to the above example). This way we would have much less results for the first join and therefore much less recursive calls when resolving the second variable. It depends on the query whether it is better to check the bound context variable at the beginning or at the end. Therefore we could use the count values as usual. With that extension we would have four additional index files which makes 20 overall.

c	?s	<p>	?o	?c	pc1
	?s	<p>	<o>	?c	poc2
	<s>	<p>	?o	?c	psc2
	<s>	<p>	<o>	?c	spoc3

Table 6.1: Index files needed when resolving variables in spo order.

The problem is that these additional indexes would just be useful for bound context variables. As soon as we want to be able to resolve unbound context variables at the beginning we need a more complex indexing scheme. That is quite difficult because when resolving a variable first, the other variables can all be either bound or unbound. This leads to the fact that many additional indexes are required.

A solution for that issue could be to choose just the indexes that would be used the most. Since many queries have bound predicate variables we could create only the indexes needed when the predicate variable is bound. Then the Engine would have to check first whether there are unbound predicate variables or not. If there are, the unbound context variables would be resolve at the end, as before. If there are not, it could be decided whether to resolve the unbound context variables at the beginning or at the end depending on the count values.

The index files that would fulfil that requirement are illustrated in table 6.1. It shows the four possible combinations when the predicate is bound and the context is unbound. For the first combination we would need a pc1 index. This one returns the context values for a given predicate value. For the count values we always need from the same index the one lower level. In this case it is the pc0 index file. In the second combination we need a poc2 (or opc2) index file because the predicate and the object variables are bound. The count values for that case are gotten from poc1. In the third combination the subject is bound instead of the object. The fourth combination is the one for which we already have the index, namely the one with all triple variables bound. With one of the indexes introduced to handle bound context variables (cs, cp, co) we could even cover a case with an unbound predicate variable. Namely the case where all four variables are unbound. We could thereto use the cs0, the cp0 or the co0 file.

So there would be three more index files for getting context values (pc1, poc2, psc2) and three more index files for getting the counts (pc0, poc1, psc1). This makes 26 index files which is quite a lot. It is difficult to say whether the speed up of the query processing is worth the slow down of the data loading. This approach and/or similar ones might be considered for future work, because resolving the context variable always at the end can cause performance issues.

Bibliography

[1] Dave Beckett. Rasqal rdf query library, 2003.

[2] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10, 2004.

[3] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[4] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[5] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.98. *Namespace Document*, 9, 2010.

[6] Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web*, pages 613–622. ACM, 2005.

[7] Graham Klyne, Jeremy J Carroll, and Brian McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[8] FAL Labs. Tokyo cabinet: a modern implementation of dbm, 2006.

[9] Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[10] Jack Rusher. Triple store. *http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html*, 2003.

[11] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

A Appendix

```
PREFIX data:
       <http://localhost:8890/>
PREFIX foaf:
       <http://xmlns.com/foaf/0.1/>

SELECT ?m
WHERE
  {
  GRAPH data:ns#graph2
     {
     ?s foaf:knows ?o.
     ?s foaf:name ?m.
     ?o foaf:name "Chris Bizer".
     }
  }

SELECT ?m
WHERE
  {
  GRAPH data:ns#graph1
     {
     ?s foaf:knows ?o.
     ?s foaf:name ?m.
     ?o foaf:name "Chris Bizer".
     }
  }

SELECT ?m
WHERE
  {
  ?s foaf:knows ?o.
  ?s foaf:name ?m.
  GRAPH data:ns#graph2
     {
     ?o foaf:name "Chris Bizer".
     }
  }
```

```
SELECT ?s
WHERE
  {
  ?s foaf:knows ?o.
  ?s foaf:name ?m.
  GRAPH ?s
     {
     ?o foaf:name "Chris Bizer".
     }
  }

SELECT ?s
WHERE
  {
  ?s foaf:knows ?o.
  ?s foaf:name ?m.
  GRAPH ?s
     {
     ?o foaf:name "Libby Miller".
     }
  }

SELECT ?s
WHERE
  {
  ?s foaf:knows ?o.
  GRAPH ?s
     {
     ?s foaf:name ?m.
     ?o foaf:name "Chris Bizer".
     }
  }

SELECT ?s
WHERE
  {
  ?s foaf:knows ?o.
  GRAPH data:ns#graph2
```

```
      {
      ?s foaf:name ?m.
      }
   GRAPH ?s
      {
      ?o foaf:name "Chris Bizer".
      }
   }

SELECT ?s
WHERE
   {
   ?s foaf:knows ?o.
   GRAPH data:ns#graph2
      {
      ?s foaf:name ?m.
     GRAPH ?s
         {
         ?o foaf:name "Chris Bizer".
         }
      }
   }

SELECT ?g
WHERE
   {
   GRAPH ?g
      {
      ?s foaf:knows ?o.
      ?s foaf:name ?m.
      ?o foaf:name "Chris Bizer".
      }
   }

SELECT ?g
WHERE
   {
   ?s foaf:knows ?o.
   ?s foaf:name ?m.
   GRAPH ?g
      {
      ?o foaf:name "Chris Bizer".
      }
   }

SELECT ?g ?f
WHERE
   {
   GRAPH ?g
      {
      ?s foaf:knows ?o.
      }
   GRAPH ?f
      {
      ?s foaf:name ?m.
      }
   ?o foaf:name "Chris Bizer".
   }

SELECT ?g ?f
WHERE
   {
   GRAPH ?g
      {
      <http://www.bizer.de#chris>
      foaf:name "Chris Bizer".
      }
   GRAPH ?f
      {
      <http://berrueta.net#me>
      foaf:name "Diego Berrueta".
      }
   }

SELECT ?p
WHERE
   {
   ?g ?p ?f
   GRAPH ?g
      {
      <http://www.bizer.de#chris>
      foaf:name "Chris Bizer".
      }
   GRAPH ?f
      {
      <http://berrueta.net#me>
      foaf:name "Diego Berrueta".
      }
   }

SELECT ?m
WHERE
   {
   ?s foaf:name ?m.
   GRAPH ?p
      {
      ?s foaf:knows ?o.
      ?o foaf:name "Libby Miller".
      }
   ?g ?p ?f.
   GRAPH ?g
      {
      <http://www.bizer.de#chris>
      foaf:name "Chris Bizer".
      }
   GRAPH ?f
      {
      <http://berrueta.net#me>
```

```
    foaf:name "Diego Berrueta".
    }
  }

SELECT ?m1 ?m2 ?m3
WHERE
  {
  GRAPH data:ns#graph1
    {
    ?p1 foaf:knows ?p2.
    }
  GRAPH data:ns#graph2
    {
    ?p2 foaf:knows ?p3.
    }
  GRAPH data:ns#graph3
    {
    ?p3 foaf:knows ?p1.
    }
  ?p1 foaf:name ?m1.
  ?p2 foaf:name ?m2.
  ?p3 foaf:name ?m3.
  }

SELECT ?m1 ?m2 ?m3
WHERE
  {
  GRAPH data:ns#graph1
    {
    ?p1 foaf:knows ?p2.
    GRAPH data:ns#graph2
      {
      ?p2 foaf:knows ?p3.
      GRAPH data:ns#graph3
        {
        ?p3 foaf:knows ?p1.
        }
      }
    }
  ?p1 foaf:name ?m1.
  ?p2 foaf:name ?m2.
  ?p3 foaf:name ?m3.
  }

SELECT ?m1 ?m2 ?m3
WHERE
  {
  GRAPH data:ns#graph2
    {
    ?p1 foaf:knows ?p2.
    }
  GRAPH data:ns#graph2
    {
    ?p2 foaf:knows ?p3.
    }
  GRAPH data:ns#graph3
    {
    ?p3 foaf:knows ?p1.
    }
  ?p1 foaf:name ?m1.
  ?p2 foaf:name ?m2.
  ?p3 foaf:name ?m3.
  }

SELECT ?m1 ?m2 ?m3
WHERE
  {
  GRAPH data:ns#graph2
    {
    ?p1 foaf:knows ?p2.
    GRAPH data:ns#graph2
      {
      ?p2 foaf:knows ?p3.
      GRAPH data:ns#graph3
        {
        ?p3 foaf:knows ?p1.
        }
      }
    }
  ?p1 foaf:name ?m1.
  ?p2 foaf:name ?m2.
  ?p3 foaf:name ?m3.
  }

SELECT ?g ?f ?h
WHERE
  {
  GRAPH ?g
    {
    ?p1 foaf:knows ?p2.
    }
  GRAPH ?f
    {
    ?p2 foaf:knows ?p3.
    }
  GRAPH ?h
    {
    ?p3 foaf:knows ?p1.
    }
  ?p1 foaf:name ?m1.
  ?p2 foaf:name ?m2.
  ?p3 foaf:name ?m3.
  }

SELECT ?g ?f ?h
WHERE
  {
  GRAPH ?g
```

```
      {
      ?p1 foaf:knows ?p2.
      GRAPH ?f
         {
         ?p2 foaf:knows ?p3.
         GRAPH ?h
            {
            ?p3 foaf:knows ?p1.
            }
         }
      }
   ?p1 foaf:name ?m1.
   ?p2 foaf:name ?m2.
   ?p3 foaf:name ?m3.
   }

SELECT ?m
WHERE
   {
   ?s foaf:name ?m.
   GRAPH ?p
      {
      ?s foaf:knows ?o.
      ?o foaf:name "Libby Miller".
      GRAPH data:ns#graph1
         {
         ?g ?p ?f.
         }
      GRAPH ?g
         {
         <http://www.bizer.de#chris>
         foaf:name "Chris Bizer".
         }
      GRAPH ?f
         {
         <http://berrueta.net#me>
         foaf:name "Diego Berrueta".
         }
   }

SELECT ?s
WHERE
   {
   ?s foaf:knows
   <http://www.bizer.de#chris>.
   GRAPH ?s
      {
      <http://www.bizer.de#chris>
      foaf:name "Chris Rod Bizer".
      }
   }
```

Listing A.1: All test queries that were used to verify named graph support. The prefixes occur only in the first query but count for every one.

List of Figures

List of Tables

List of Listings