Visual Query Formulation and Subclass Reasoning for Linked Open Data

Simen Heggestøyl, master thesis spring 2014
Visual Query Formulation and Subclass Reasoning for Linked Open Data

Simen Heggestøyl

May 2, 2014
Abstract

The cloud of Linked Open Data is steadily growing, yet it remains largely inaccessible to the general public, due to the technical barrier posed by the requirement to know formal query languages. In this thesis, we present new approaches for visual query formulation toward arbitrary SPARQL endpoints. Class hierarchies appear naturally in many such datasets. To support these cases, we assess different approaches for supporting subclass reasoning in SPARQL queries, and show how subclass hierarchies can be integrated in visual search tools.

We present five different alternatives for supporting subclass reasoning: RDFS backward reasoning, RDFS forward reasoning, property paths, query rewriting, and query federation. The technical details of our suggested approaches for supporting subclass reasoning are thoroughly discussed with a case study drawn from the Norwegian Entity Registry. The performance of each approach is measured in a benchmarking experiment, where RDFS reasoning and query rewriting is found to perform well, while the property paths and query federation approaches are found to be inadequate for supporting subclass reasoning.

We present two visual search tool prototypes for Linked Open Data, developed to improve the accessibility to Linked Open Data sources for casual users, and we show how subclass hierarchies can be integrated in these tools. The usability of our suggested subclass integration is assessed in a user study, where it is shown to perform adequately.
Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, Martin Giese and Guillermo Vega-Gorgojo, for excellent guidance during my thesis work. I would also like to thank Kjetil Kjernsmo for fruitful discussions.

This thesis would never exist, hadn’t it been for the camaraderie of the 8th floor gang. Thank you Kjetil Andresen, Henrik Bjørlo, Peter Brottveit Bock, Kyrre Havik Eriksen, Erlend Kristiansen, Håkon Normann, Simen Skogly Russnes, Daniel Rødskog, and Patrick Bauge Skevik. A special thanks goes to Martin Harbu Bielecki for proofreading the thesis.

I would like to thank Petter Berg Grav, Ngoc Nguyen, Martin Stensgård, and Lars Tveito, for keeping my spirits high throughout the semesters.

Lastly, I would like to thank my family for their endless support: mom, dad, Anders, Åge, and Emma.

Thank You
Simen Heggestøyl
Contents

1 Introduction ............................................. 1

2 Background ............................................. 5
   2.1 Semantic Web Technologies ......................... 5
   2.2 Linked Open Data .................................. 7
   2.3 The Norwegian Entity Registry ..................... 9
   2.4 Class Hierarchies .................................. 9
   2.5 Summary ........................................... 12

3 Goals ................................................... 13

4 Methods ............................................... 15
   4.1 Quantitative Analysis ............................... 15
   4.2 Qualitative Analysis ............................... 16

5 Overview of Subclass Reasoning Approaches for Linked Data ......................... 19
   5.1 RDFS Backward Reasoning ......................... 19
   5.2 RDFS Forward Reasoning ......................... 21
   5.3 Property Paths .................................... 22
   5.4 Query Rewriting .................................. 23
   5.5 Query Federation .................................. 25
   5.6 Summary ........................................... 25

6 Implementing Subclass Reasoning for Linked Data ................................... 27
   6.1 Modeling the NACE Hierarchy in RDF ............... 27
   6.2 RDFS Backward Reasoning ......................... 29
   6.3 RDFS Forward Reasoning ......................... 29
   6.4 Property Paths .................................... 30
   6.5 Query Rewriting .................................. 30
   6.6 Query Federation .................................. 32
   6.7 Summary ........................................... 33

7 Evaluation of Subclass Reasoning Approaches ................................... 35
   7.1 Performance Analysis ............................... 35
   7.2 Qualitative Comparison ............................. 45
   7.3 Summary ........................................... 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Visual representation of the statement “The University of Oslo is in the country of Norway”</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Visual representation of the graph from Section 2.4.1</td>
<td>11</td>
</tr>
<tr>
<td>8.1</td>
<td>Usability and expressivity of approaches for querying Linked Data (derived from table in [Veg+14])</td>
<td>50</td>
</tr>
<tr>
<td>8.2</td>
<td>Using PepeSearch to search for organizations in the Semicolon dataset</td>
<td>53</td>
</tr>
<tr>
<td>8.3</td>
<td>Results after querying for organizations founded between year 1790 and 1800</td>
<td>54</td>
</tr>
<tr>
<td>8.4</td>
<td>Advanced version of the search tool prototype</td>
<td>55</td>
</tr>
<tr>
<td>8.5</td>
<td>Architecture of PepeSearch (from [Veg+14])</td>
<td>56</td>
</tr>
<tr>
<td>8.6</td>
<td>Using the advanced search tool to search for authors of Semantic Web papers</td>
<td>58</td>
</tr>
<tr>
<td>8.7</td>
<td>Using PepeSearch to search for wellbore oil samples</td>
<td>59</td>
</tr>
<tr>
<td>8.8</td>
<td>Browsing subclasses of the education sector</td>
<td>60</td>
</tr>
<tr>
<td>9.1</td>
<td>Answers to question 1 and 2 in the subclass interface survey</td>
<td>64</td>
</tr>
<tr>
<td>9.2</td>
<td>ANOVA result plots</td>
<td>68</td>
</tr>
<tr>
<td>10.1</td>
<td>Tiny, but successful, fishing businesses in Norway found by PepeSearch</td>
<td>71</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 NACE hierarchy for code 85.421 ........................................ 10
5.1 Overview of subclass reasoning approaches ........................................ 20
7.1 Storage space comparison ......................................................... 36
7.2 Test server specifications ........................................................ 37
7.3 Cold query response time summary for questions 1-6 (in milliseconds) ........ 40
7.4 Warm query response time summary for questions 1-6 (in milliseconds) ........ 42
7.5 Total cold response time means ................................................... 44
7.6 Total warm response time means in comparison to the cold response time means .... 44
7.7 Qualitative assessment of subclass reasoning approaches ................. 45
9.1 Task orderings forming a Latin square ........................................ 62
9.2 Selected free-text answers to the question “What did you dislike more about this tool [PepeSearch]?” ........... 63
9.3 Selected free-text answers to the question “How would you improve this tool [PepeSearch]?” ........... 63
9.4 Free-text answers to the question “Other comments about the sector hierarchy” ........... 65
9.5 ANOVA table for performance on questions where subclass reasoning was relevant .... 67
A.1 Table of common RDF prefixes .............................................. 75
Chapter 1

Introduction

Imagine the following situation: an ambitious journalist is in the process of writing an article about tiny, but successful, fishing businesses in Norway. For his research, he would like to know the answer to the following question:

Which Norwegian fishing companies have ever had a yearly revenue of more than 100,000,000 (one hundred million) Norwegian kroner, but no more than 10 employees?

To obtain this information, our journalist has some options:

1. He may go ahead and do some research. After a few basic web searches, it seems that the answer to this question is not readily available on the web.

For instance, a web search at Google with the query string “small fishing companies in Norway” turns up a list of fishing companies in Norway, but information on their size or revenue is lacking. The journalist will have to look up this additional information from other sources, and manually filter out the companies that don’t fit the problem description. Unfortunately, this is a very demanding and time consuming task. Furthermore, such manual information gathering is prone to errors and incomplete results, especially if he doesn’t find any authoritative information sources. That is, how can he be sure that the list of fishing companies he found through Google is complete?

2. In this particular case, an authoritative source for information about companies in Norway is the Norwegian Entity Registry (Norwegian: Enhetsregisteret), as published by The Brønnøysund Register Centre (Norwegian: Brønnøysundregistrene). Using this data source gives the journalist increased confidence in the
completeness and correctness of the data, compared to the web search approach.

But there is one major shortcoming with this approach as well, in that it only supports look-up based on the company name or organization number, so he would need to know any of those beforehand.

3. A third option would be to query a version of the Norwegian Entity Registry that has been published as Linked Open Data (LOD). A detailed explanation of the term Linked Open Data will be given in Section 2.2. In short, it means that the data is freely available for use and reuse, and that it’s published according to the Linked Data principles [Ber06], meaning that the data is published in a special format making it possible to interlink it with other datasets.

This possibility exists due to the efforts of Semicolon, an ongoing research project that will be elaborated on in Section 2.3.

The query code he would need to write in order to query this dataset would look something like this:

```sparql
SELECT DISTINCT ?companyName WHERE {
  ?company a org:Enhet ; rdfs:label ?companyName ;
  org:nacekode inace:A3.1 ;
  org:antAnsattePåDato [ a org:AntAnsattePåDato ;
  [ a reg:Regnskap ;
    ?enhet ?company ;
    reg:Årsresultat ?revenue ]
  FILTER (?employees <= 10 && xsd:integer(?revenue) > 100000000)
}
```

This approach is good because he will get an answer within minutes. He can also be confident in the data source, as it is an authoritative source for information about Norwegian companies. Furthermore, the answer will be precise, as the above snippet is written in a formal query language called SPARQL [HS13]. A formal language has formally defined syntax and semantics. In this case, it also means that running the same code over again would produce the same result, given that the data source hasn’t changed. This is what is meant by a precise answer.

But it also means that the journalist has to know the SPARQL language to have any use of the data source. The journalist is not a computer expert, so he realizes he will have to hire somebody to write these codes for him. This process may not only be time consuming, but possibly expensive as well.

Similar scenarios involving other Linked Open Data sources can be imagined:
• The historian exploring Project Gutenberg’s vast collection of books.
• The music journalist collecting music facts from DBTune.
• The eager student browsing DBpedia, which is a Linked Open Data collection of facts from Wikipedia.

These use cases are only a few examples of how people can benefit from access to the cloud of Linked Open Data that has been growing rapidly since 2007 [HB11, Ch. 3.1]. Governments around the world are publishing their public data as Linked Open Data, and the Norwegian Entity Registry is an example of such governmental data. Governments are in fact the greatest contributor to the Linked Open Data cloud in terms of sheer data size [BJC11].

How would the situation look if we were able to eliminate the need for the computer expert in these situations? That is, if the “man on the street” were able to exploit Linked Open Data sources, without having to learn a formal query language?

We can see that people would benefit from having a flexible and powerful, yet simple and unassuming method of exploiting Linked Open Data. Appropriate tools and interfaces are needed to fill the gap between the competence of the users and the formal query formulation process, as requested in [Sha+12].

Several natural language interfaces have been developed for this purpose, such as Querix [KBZ06], FREyA [DAC11], and AutoSPARQL [SN12]. However, such tools inherit one of the big problems of natural language: ambiguity [KBZ06].

Graphical user interfaces for querying Linked Open Data sources are sparser in the literature. One such interface, Semantic Crystal, was compared with three natural language interfaces in [KB07]. It was, however, found to perform worst of them all, both with respect to query formulation time, result accuracy, and user likability. The study concluded that the graphical user interface was too complex for the target users.

The group for Logic and Intelligent Data (LogID) at the Department of Informatics, University of Oslo, wants to have another stab at this problem. We are researching on graphical, interactive interfaces to enable the general public to exploit Linked Open Data sources, and in particular a Linked Open Data source published by the Norwegian government.

http://wifo5-03.informatik.uni-mannheim.de/gutendata/
http://dbtune.org/
http://dbpedia.org/
https://www.wikipedia.org/
http://www.mn.uio.no/ifi/forskning/grupper/logid/
In this master thesis, the focus will be on how we best can support class hierarchies in the tool. Class hierarchies appear naturally in many contexts, and we want to support these cases. This problem is twofold:

1. The interface of the search tool must present class hierarchies found in the dataset in an intuitive way, matching the way humans naturally think about class hierarchies.

2. The tool has to be aware of these hierarchies in order to correctly formulate queries involving them. It is not obvious how this should be done, and multiple alternatives will have to be explored.

The remainder of this document is structured as follows: Chapter 2 presents topics that the reader has to be aware of before a meaningful problem description can be given. Chapter 3 provides a concise problem description. Chapter 4 describes the methods we employed when trying to achieve our goals. Chapter 5 presents different approaches for supporting subclass reasoning for Linked Data. Chapter 6 presents a case study where each of the subclass reasoning approaches are applied to the Norwegian Entity Registry dataset from the Semicolon project, while Chapter 7 gives an evaluation of all the proposed subclass reasoning approaches. Chapter 8 introduces a tool prototype for searching Linked Open Data. Chapter 9 gives an evaluation of the subclass user interface made for the prototype. Lastly, Chapter 10 summarizes the results and contributions of this work.
Chapter 2

Background

This chapter gives a brief introduction to semantic technologies; establishes the terms Linked Data and open data; provides a recap of the Semicolon research project; and presents a particular governmental Linked Open Data source for the Norwegian Entity Registry.

2.1 Semantic Web Technologies

The existing World Wide Web works very well for unstructured document exchange, but remains largely unstructured beyond single applications. The idea of the Semantic Web is to create a web of data on top of the existing Web, making data discoverable and reusable across different applications [W3C13b].

How is this beneficial in practice? An ideal Semantic Web world would free users from having to crisscross data sources that deal with common concepts. For instance, while planning an overseas vacation, you would combine (at least) two data sources: your calendar and the airline flight schedule. Today, you would have to do this manually; the applications cannot do this for you, because they are working with separate concepts of time in their own data formats. Semantic Web technology tries to solve this problem, by providing a common data framework [W3C13b].

This section gives a brief introduction to core Semantic Web technologies.

2.1.1 Data Representation

Skipping some technical details about syntax and serialization, we can consider an RDF triple to be the most basic building block of the Semantic Web world. An RDF triple consists of three parts: a subject,
A triple of resources represents a statement. For instance, we might want to express that the University of Oslo is located in the country of Norway. Expressed in RDF, the subject of the triple would represent the University of Oslo, the predicate would represent an “in country” relation, and the object would represent the country of Norway:

\[ \text{University of Oslo} \quad \text{in country} \quad \text{Norway} \].

A collection of RDF triples is called an RDF graph. It is often convenient to illustrate RDF graphs as a collection of nodes representing the subjects and objects, with directed arcs between them, representing the predicates. Figure 2.1 illustrates the previous statement, interpreted as a single-triple RDF graph.

![Figure 2.1: Visual representation of the statement “The University of Oslo is in the country of Norway”](image)

RDF triples provide a natural model for expressing statements. Natural language strings are however inherently ambiguous. For instance, the string “Norway” may refer to the country of Norway, as well as the Iowa city with the same name. RDF resolves this ambiguity issue by having uniform resource identifiers (URIs) as resource names. An URI is a string intended to unambiguously identify a resource. In this manner, we may use the URI \text{http://dbpedia.org/resource/Norway} to explicitly refer to the country of Norway, and \text{http://dbpedia.org/resource/Norway,_Iowa} to refer to the city [HKR10, Ch. 2].

It is often convenient to shorten URIs by declaring prefixes, making the URIs more readable. For instance, we might want to declare the prefix \text{dbpedia} for the URI stem \text{http://dbpedia.org/resource/}, allowing us to write \text{dbpedia:Norway} for the URI of the country of Norway.

For the remainder of this thesis, we will use URI prefixes wherever applicable to make RDF statements more readable.Whenever it’s insignificant what the prefix should be, we will use an empty prefix (e.g. :Norway). Otherwise, we will use one of the prefixes listed in Appendix A: Prefix Table. And lastly, the resource \text{rdf:type} will be used interchangeably with its shorthand notation: a.

### 2.1.2 Querying RDF Graphs

We have seen that we can represent statements as triples in an RDF graph. So far, so good, but how are we actually going to access this data
afterwards?

Luckily, such a representation lends itself naturally to queries by
graph pattern matching. The SPARQL query language, showcased
briefly in Chapter [1] enables us to query RDF graphs by matching it
against constructed graph patterns [HKR10 Ch. 7]. Servers that accept
SPARQL queries and return the results are called SPARQL endpoints.

Rather than getting into the details of the SPARQL syntax (see
[HS13] for that), we want to give an idea of how graph pattern matching
is done by means of a simple example. Consider the following RDF
graph of three triples:

```
```

Imagine that we are interested in retrieving a list of every university
known to lie in the country of Norway from this dataset. We achieve
this by constructing a new graph of one triple with the :inCountry
relation in the predicate position, and :Norway in the object position,
but with a variable in the subject position. This variable acts as a wild
card; it matches any resource as long as the predicate and object match
the fixed ones. In the following query graph, the variable is arbitrarily
named ?university:

```
```

This query yields :UniversityOfOslo and :NTNU as possible
instantiations of ?university, which answers our question.

2.2 Linked Open Data

We use the term Linked Open Data (or LOD for short) to describe
pieces of data that are both linked and open. Open data and Linked
Data are independent concepts, and will be described separately in the
following two subsections. Then follows a presentation of an edition of
the Norwegian Entity Registry that is published as Linked Open Data.

2.2.1 Open Data

Following the definition by the Advisory Council of the Open Definition,
the definition of “open data” can be summarized as follows:

“A piece of data or content is open if anyone is free to use,
reuse, and redistribute it — subject only, at most, to the
requirement to attribute and/or share-alike.” [Cou]
A requirement to attribute means that one would have to credit the data owner when applying the data. A requirement to share-alike means that when the data is reused, it has to be licensed under the same share-alike conditions as the original one, ensuring that the data remains open.

Many people feel that data from the public sector should be licensed as open data. Håkon Wium Lie is one such proponent. He argues on the grounds of “fairness”; when we, as taxpayers, collectively fund the production of some data, it is simply fair that this data is given back to us. Wium Lie draws The Norwegian Mapping Authority’s nautical charts as an example. The production of the charts is largely funded by tax money. Still, private persons have to pay approximately 10,000 Norwegian kroner to acquire electronic copies of these charts. Wium Lie suggests that the citizens are charged twice – once through tax, and once through direct payment.[Lie09; HR13]

The Norwegian government declared the following in their political declaration for 2009-2013:

“The Government shall: see to it that information of public interest as a rule should be free, accessible and available for everyone in digital form” [PPP09]

2.2.2 Linked Data

Linked Data denotes a way of publishing data so that it becomes possible to interlink the data with other data sources [HB11; Ch. 2]. This puts the data in a greater context, increasing the value of all the interlinked data. It also enables machines to explore the data and extract meaning through the links – much like we humans explore the World Wide Web through hyperlinks.

Tim Berners-Lee has devised a four-step recipe [Ber06] on how to transform your ordinary data into Linked Data:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using RDF or SPARQL.
4. Include links to other URIs, so that they can discover more things.

To have Linked Data published in this way is crucial to the mission of the Semantic Web, as it provides a standardized way of interlinking datasets. We will however see that such a Linked Data representation becomes useful even when applied in a limited domain.

1Norwegian politician and inventor of Cascading Style Sheets (CSS)
2Best known as the inventor of the World Wide Web
2.3 The Norwegian Entity Registry

The Semicolon project is an ongoing research project led by Karde AS, with support from a number of large Norwegian actors, including Computas, The Brønnøysund Register Centre, and the University of Oslo to improve interoperability in the public sector, but also between the public sector and the citizens and businesses of Norway [EM11].

As one of the outcomes of this project, the Norwegian Entity Registry (Norwegian: Enhetsregisteret) and the Norwegian Accounting Registry (Norwegian: Regnskapsregisteret) have been published as Linked Open Data in the RDF format.

The Entity Registry contains information about every Norwegian registered “entity”. An “entity” can – for instance – be a company or a foundation [Brøa]. The registry includes over 350,000 such entities. Every entity is linked to relevant information, such as its organization number, date of founding, number of employees, or daughter entities. Linking together data from the Entity Registry and the Accounting Registry makes it possible to retrieve historical accounting information for Norwegian entities as well.

Before the Semicolon project, accessing this data was inconvenient, as explained in Chapter[1]. The Brønnøysund Register Centre maintains a website[4] which supports per-company look-up, when the name of the company is already known. Some limited subsets of the company registry have also been published as structured data[5]. After being published as Linked Open Data, the data can be explored through a web interface[6] or queried through a SPARQL endpoint[7]. But as previously mentioned, this may still not be very useful to the average citizen.

We will use this dataset as our prime case study while developing a graphical user interface to simplify exploration and exploitation of Linked Open Data sets.

2.4 Class Hierarchies

Class hierarchies show up naturally in many contexts. In the case of the Norwegian Entity Registry, one such hierarchy is the classification of companies into sectors.

These sectors are arranged by the European Union (EU) into a five level deep hierarchy called the NACE hierarchy [Com10]. At each level

---

[1] For a complete list see: http://www.semicolon.no/?page_id=154
http://brreg.no/
http://data.norge.no/organization/registerenheten-i-brønnøysund
http://data.computas.com/
http://data.computas.com:3030/sparql
in the hierarchy, a sector is further refined from its parent sector. This results in broad sectors at the top level, such as “Manufacturing” or “Mining and quarrying”, and in highly specialized sectors at the bottom level, such as “Manufacture of bricks, tiles and construction products, in baked clay” or “Quarrying of ornamental and building stone, limestone, gypsum, chalk and slate”.

For instance, the University of Oslo’s NACE code is 85.421, which means it belongs to the bottom level class “University education”. But it also means that it belongs to the top-level class “Education”, and all the other subclasses in between (listed in Table 2.1).

<table>
<thead>
<tr>
<th>Level</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P</td>
<td>Education</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
<td>Education</td>
</tr>
<tr>
<td>3</td>
<td>85.4</td>
<td>Higher education</td>
</tr>
<tr>
<td>4</td>
<td>85.42</td>
<td>Tertiary education</td>
</tr>
<tr>
<td>5</td>
<td>85.421</td>
<td>University education</td>
</tr>
</tbody>
</table>

### 2.4.1 Why Subclass Reasoning Is Needed

Consider the following RDF graph (visualized in Figure 2.2 on page 11, where blue nodes represent instances, and yellow nodes represent classes):

```
:MarineFishing rdfs:subClassOf :Fishing .
:FreshwaterFishing rdfs:subClassOf :Fishing .
:marineFishing a :MarineFishing .
:freshwaterFishing a :FreshwaterFishing .
:hansenFiskAS :hasSector :marineFishing .
:fjellfeskSA :hasSector :freshwaterFishing .
```

Imagine that a user is interested in retrieving every company that is part of the fishing sector. From the above graph, we see that the fishing sector would include the freshwater fishing sector, and the marine fishing sector, which Fjellfesk SA and Hansen Fisk AS are respective members of. The user might therefore expect Fjellfesk SA and Hansen Fisk AS to be members of the fishing sector as well. We will however see, that without further measures, this will not be the case.

Consider the following query, asking for every company that is part of the fishing sector:

```sql
SELECT ?company WHERE {
  ?company rdfs:subClassOf ?fishering .
}
```
Both :MarineFishing and :FreshwaterFishing are subclasses of the :Fishing sector, so we might expect \( ?\text{company} \) to be instantiated as :hansenFiskAS and :fjellfeskSA. This is not the case with pure graph matching though, as the matching algorithm remains unaware of our intended semantic meaning behind the rdfs:subClassOf property. The result of this query will therefore be empty.

We would rather have the query above to actually yield :hansenFiskAS and :fjellfeskSA as results, like we would expect intuitively. To achieve this, we need to build additional logic into our tool. We will call this additional logic subclass reasoning. Chapter 5 presents several proposals on how subclass reasoning can be supported.
For the remainder of this document, we will use the word “reasoner” for a program that is able apply rules to infer new facts from a given dataset [Heb+09, Ch. 8]. In particular, we will work with reasoners that are able to infer new RDF triples from a set of rules and a given RDF graph. Subclass reasoning is one type of such inference.

2.5 Summary

We have seen that large amounts of data reside in the growing cloud of Linked Open Data. The data is stored in the RDF format, and collected in RDF graphs. When these RDF graphs are served through SPARQL endpoints, they become accessible to those who know the SPARQL formal query language.

The requirement to know a formal query language in order to exploit the data creates a barrier for non-technical users. We see that there is a need for accessible tools granting also non-technical users the power to exploit Linked Open Data sources.

Many Linked Open Data sets, such as the Norwegian Entity Registry, contain class hierarchies. We want to support these cases as well, and it is therefore important to build support for reasoning over such class hierarchies into our tools.

With that, we are ready to state our concrete goals in the next chapter.
Chapter 3

Goals

The LogID group is researching on how to make graphical user interfaces to enable end-users to explore and exploit Linked Open Data sources, such as the Norwegian Entity Registry, without requiring knowledge of formal languages. A visual query formulation prototype is already in development as part of an ongoing EU project called Optique \cite{Soy+13a, Soy+13b}. We will use this prototype as a base for our own.

The goals for this master thesis are the following:

1. Analyze different alternatives for supporting subclass reasoning in query formulation.

2. Compare the different alternatives, and propose an approach for supporting subclass reasoning in a search tool prototype.

3. Propose and integrate user interface mechanisms for supporting class hierarchies in a search tool prototype.

4. Apply the search tool prototype to the case of the Norwegian Entity Registry.
Chapter 4

Methods

This chapter presents the methods we have employed for data collection and analysis during the work for this thesis. As an overarching methodology, we have followed the steps of the engineering method (adapted from [Adr92]):

1. Observe existing solutions
2. Propose better solutions
3. Develop the proposals
4. Measure and analyze

This pattern was followed both during our approach to the subclass reasoning problem, and to the problem of integrating subclass hierarchies in our search tool prototype.

As for the discussion of our concrete methods, we have split it into two main parts: quantitative methods and qualitative methods. These parts are elaborated on in the sections that follow.

4.1 Quantitative Analysis

In order to fulfill our goal of supporting subclass reasoning in visual query formulation, we need to investigate the different approaches we can take. After investigating the alternatives, each approach will be implemented in order to evaluate the performance of each approach.

In quantitative research, one is concerned with numerical observations, and how one can derive meaning from them [SGB05]. In this part of our analysis, we want to evaluate the performance of our subclass reasoning approaches numerically, that is, we want to assign numbers to each approach that describes some quality about them.
Three techniques can be employed in order to evaluate the performance: analytical modeling, simulation, and measurement [Jai91, Ch. 3.1].

Analytical modeling involves the creation of mathematical models of the approaches, and analyzing the implications of those models. This analysis can be performed without a working system, or even a prototype. It is often unfeasible to take every possible parameter into account in this approach, so simplifications have to be made, which in turn often leads to imprecise results [Jai91, Ch. 3.1]. Additionally, the available tools for pure analytical modeling are in many cases too weak. For instance, in calculus, most integrals cannot be solved analytically [Mør13]. In such cases, we must resort to simulation, which is discussed next.

As with analytical modeling, simulation can be performed without actually having a running system; the goal is simulate a real working system. Having a computer simulation makes it easy to vary the different parameters, and to measure the effect they have on the performance of the (simulated) system. It typically provides a higher accuracy than what analytical modeling can provide [Jai91, Ch. 3.1].

Making measurements is the most costly of the techniques. It requires a running system, or a prototype of the system we want to evaluate. The system has to be set up prior to measuring, including needed hardware and measurement software. Unforeseen difficulties can add to the setup time, and make this a very time-consuming task. On the bright side, when conducted correctly, measurements of a real, running system can potentially provide highly accurate numbers [Jai91, Ch. 3.1].

We chose to make measurements when evaluating the performance of our proposed subclass reasoning approaches in Section 7.1. The choice was motivated by the fact that we had access to all needed equipment, and a working prototype ready at our hands. We also had the time to write custom measurement software, which laid the ground for what we believe to be a solid evaluation setup. Contrary to simulation, a drawback of the measurement technique is that the different parameters are hard to vary. This is not critical in our case study, as we will lock every parameter except one, namely the subclass reasoning approach in use by the system.

### 4.2 Qualitative Analysis

In qualitative research, one is concerned with data in the form of words rather than numbers. In Section 7.2, we compare the proposed subclass reasoning approaches with respect to non-discrete metrics such as “perceived complexity” and “setup effort”. Qualitative methods are
better suited in this situation than purely quantitative ones, as the textual nature of the data is lost when trying to capture such metrics numerically [KM05]. For instance, “setup effort” could be measured by the number of minutes it took for us to set up a particular subclass reasoning approach, but it seems more valuable to us to have a textual description of our impression of the setup effort.

One of our goals was to propose and integrate user interface mechanisms for supporting class hierarchies in a search tool prototype. After presenting our proposed solution in Chapter 8, we evaluated our proposal by means of a user experiment. The design of this experiment is elaborated on in Chapter 9.

As the experiment involved measuring impressions of human participants, we once again felt the need for qualitative methods to deal with the human complexity. Qualitative user data was collected from the experiment through a questionnaire, both in form of free-text answers, and through Likert scale ratings. With Likert scales, the users are asked to select a point on a qualitative scale, for instance ranging from “Very unconfident” too “Very confident”, or from “Strongly disagree” to “Strongly agree”. The Likert scales we employed in our questionnaire were divided into 7 discrete points, as an odd number of points makes it clear to the user which is the neutral choice [Hea09, Ch. 2].
Chapter 5

Overview of Subclass Reasoning Approaches for Linked Data

We saw in Section 2.4.1 that subclass reasoning is needed to correctly answer Linked Data queries involving class hierarchies, such as the NACE sector hierarchy in the Norwegian Entity Registry. Subclass reasoning can be supported in a number of different ways, and several alternatives will be implemented and evaluated.

This chapter presents five different ideas for approaches on how to support subclass reasoning for Linked Data. The descriptions are conceptual; a case study on how these approaches can be implemented in practice is given in Chapter 6.

Table 5.1 provides a convenient overview of our suggested approaches for supporting subclass reasoning. Other possibilities may exist, but these approaches are, to the breadth of our knowledge, common ways of supporting inference [HKR10, Ch. 3] [HS13, Ch. 9] [JJB09] [W3C13a].

Three of the approaches may require server access (marked “Maybe” in the table). Server access is only needed if the subclass hierarchies aren’t already present in the dataset. This is for instance the case for the NACE class hierarchy of the Norwegian Entity Registry.

5.1 RDFS Backward Reasoning

The most natural approach for supporting subclass reasoning would perhaps be to use RDF’s own vocabulary description language: RDF Schema (or RDFS for short) [W3C14a].

RDFS provides mechanisms for structuring RDF resources into
Table 5.1: Overview of subclass reasoning approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Client/server?</th>
<th>Need server access?</th>
<th>Additional notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward reasoning</td>
<td>Server-side</td>
<td>Maybe</td>
<td>Server requires RDFS support</td>
</tr>
<tr>
<td>Forward reasoning</td>
<td>Server-side</td>
<td>Maybe</td>
<td>Potential maintenance problems</td>
</tr>
<tr>
<td>Property paths</td>
<td>Server-side</td>
<td>Maybe</td>
<td>Requires SPARQL 1.1</td>
</tr>
<tr>
<td>Query rewriting</td>
<td>Client-side</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Query federation</td>
<td>Both</td>
<td>No</td>
<td>Requires SPARQL 1.1; need extra server</td>
</tr>
</tbody>
</table>

classes, and describing relations between them. What makes RDFS attractive for our purpose, is that is comes with formally defined semantics, meaning that we have an unambiguous way of interpreting such relations.

This allows the server to automatically infer that new triples should be regarded as part of the graph, following the RDFS entailment rules [W3C14b].

It makes sense to consider one such rule for an example. In specific, rule rdfs9\(^1\) states that whenever we have two triples on the form:

\[
X \text{ rdfs:subClassOf } Y . \\
A \text{ rdf:type } X .
\]

Where \(X\), \(Y\), and \(A\) are arbitrary resources, we add an additional triple to the dataset:

\[
A \text{ rdf:type } Y .
\]

Informally, we can interpret this as “when \(A\) is of type \(X\), and \(X\) is a subclass of \(Y\), \(A\) is also of type \(Y\)”.

Coming back to the fishing sector example from Section 2.4.1, the mentioned rule allows the server to infer two new triples:

\[
:\text{marineFishing a :Fishing .} \\
:\text{freshwaterFishing a :Fishing .}
\]

Thus, both :hansenFiskAS and :fjellfeskSA, which were members of a :marineFishing and :freshwaterFishing respectively,

\(^1\)http://www.w3.org/TR/rdf-mt/#RDFS9Rules
will be retrieved by the query:

```sparql
?company :hasSector ?sector .
?sector a :Fishing .
```

When backward reasoning algorithms are employed to support inference, the inference engine works backward from the goal, trying to infer new facts that support a given statement [RN10]. In this context, a backward reasoning strategy implies that the server will have to infer new triples on-demand when responding to a SPARQL query.

### 5.2 RDFS Forward Reasoning

Another RDFS approach is to use forward reasoning on the server instead of backward reasoning. The difference is that with forward reasoning, the server generates every triple that can possibly be inferred from the RDF graph beforehand, using the RDFS inference rules.

This approach will require more storage space, but may possibly enable more efficient query answering, as no extra per-query reasoning is required [Heb+09, p. 149].

Though it may prove less flexible, as we will have to rerun the reasoner after making changes to the ontology, in case new triples ought to be inferred. A bigger problem arises if statements are removed from the ontology. It may lead to the situation that previously inferred triples still exist in the database, even if they cannot be inferred from the new ontology [Heb+09 p. 150].

The inference might as well be performed by an external reasoner before the data is loaded into the database. This makes forward reasoning viable for RDF databases that do not support reasoning directly.

For an example of how forward reasoning works, consider the following RDF graph:

```rdfs
:MarineFishing rdfs:subClassOf :Fishing .
:Fishing rdfs:subClassOf :FishingAndAquaculture .
:hansenFiskAS :hasSector :marineFishing .
:marineFishing a :MarineFishing .
```

Performing RDFS forward reasoning on this rather small dataset would add at least five new inferred triples to the graph:

```rdfs
:Fishing a rdfs:Class .
:MarineFishing a rdfs:Class .
:FishingAndAquaculture a rdfs:Class .
```
By adding these inferred triples to the dataset, we are able to answer the following query, asking for companies belonging to the “Fishing and aquaculture” sector:

?company :hasSector ?sector .  
?sector a :FishingAndAquaculture .

This query correctly returns :hansenFiskAS as a possible instantiation of ?sector, since :marineFishing is inferred by the RDFS forward reasoner to belong to the :FishingAndAquaculture class.

Note that the RDFS forward reasoner inferred some additional triples that we didn’t need to have for the above query to work. Indeed, the graph more than doubled in size. This is the price we pay with forward reasoning in general; we cannot know beforehand which triples will be needed later, so we have to infer and store all of them, even if they are never actually used [Heb+09, p. 150].

5.3 Property Paths

Property paths were introduced in SPARQL 1.1. They allow queries to match graph paths of arbitrary length [HS13, Ch. 9]. For an ordinary triple pattern that only contains one property, the property path is of length 1.

Property paths extend the syntax of SPARQL with a path language, bearing resemblance to the notation that is often used for regular expressions.

For instance, the following query uses a property path of length 2, and would match every company that has a sector of type :Fishing:

?company :hasSector/rdf:type :Fishing .

Without use of property paths, an equivalent query would be:

?company :hasSector :someSector .  
?someSector rdf:type :Fishing .

By using property paths, we are able to pose a simple query for retrieving any resource of a given type, or of any subclass of that type.

Going back to the introductory example of this chapter, where :MarineFishing and :FreshwaterFishing were subclasses of :Fishing, the following query pattern would match all the three fishing sectors,

2http://www.w3.org/TR/xmlschema-2/#regexs
but without any reasoning being performed on the server apart from following the property path:

```reasoning
```

We can interpret the preceding query as “retrieve every sector of type :Fishing, or of a type that is a subclass of :Fishing”. Analogous to regular expressions, the star succeeding rdfs:subClassOf allows a property path of zero or more rdfs:subClassOf properties, meaning that the subclass hierarchy can be arbitrarily deep.

### 5.4 Query Rewriting

By altering queries before they are sent to the endpoint, we may be able to emulate subclass reasoning without requiring changes to be made on the server-side at all.

Returning to the fishing sector example from Section 2.4.1, recall that original query was not able to retrieve the companies in the “Fishing” sector because members of the :FreshwaterFishing and :MarineFishing classes weren’t recognized as members of the :Fishing superclass:

```reasoning
?company :hasSector ?sector .

?sector a :Fishing .
```

If, however, the client is able to detect that :Fishing is a superclass, it is possible to rewrite the query to compensate for the lack of reasoning on the server.

Several different approaches seem appropriate here. We will present three different ideas in the following subsections. Common for all the suggested approaches is that they are not limited to detection of subclasses defined by rdfs:subClassOf; we may indeed declare additional, specialized class hierarchies on the client side.

#### 5.4.1 Unions

A straightforward strategy is to split the query into a union of several graph patterns. The following pattern instantiates every company that has a sector which is :fishing or :marineFishing or :freshwaterFishing (or being inclusive):

```reasoning
?company :hasSector :fishing .

U

?company :hasSector :marineFishing .
```

23
5.4.2 Instance Filtering

By keeping variables that are subject to subclass reasoning unbound in the query, we may instead filter out inappropriate results after the graph pattern matching has been performed.

For the fishing sector example above, this strategy amounts to keeping the `?sector` variable unbound in the query, allowing instances of every sector class to be matched. In this way, we are sure that we are not missing any answers, but we may have gotten too many. A filter is then applied, keeping only the results where `?sector` is bound to one of the instances `:fishing`, `:marineFishing`, or `:freshwaterFishing`:

```reasoning
?company :hasSector ?sector .
WHERE ?sector ∈ {:fishing, :marineFishing, :freshwaterFishing}
```

5.4.3 Attribute Filtering

Instance filtering, as described in the previous subsection, requires explicit listing of every possible sector instance. When the number of instances grows large, this might become cumbersome and inefficient.

An alternative to filtering by instance is to filter by some of the instances’ attributes, in that way dividing instances into classes implicitly.

Indeed, the NACE code of a sector seems like a natural choice for such an ordering, as the NACE hierarchy already has an (almost) lexicographical order. Only the top-level classes need special consideration; these are labeled with letters instead of numerical codes. For these, numerical ranges have to be explicitly defined.

For instance, to select instances of the top-level class “Fishing”, with NACE code 03.1, we need to filter by codes in the lexicographical range ["03.1", "03.2"). That is, every sector with a code from and including 03.1, to, but not including, 03.2. This would yield both the “Fishing” sector, “Marine fishing” sector and the “Freshwater fishing” sector, with NACE codes 03.1, 03.111, and 03.120 respectively, as they are all lexicographically larger or equal to 03.1, but smaller than 03.2.

```reasoning
?company :hasSector ?sector .
WHERE "03.1" <= ?code < "03.2"
```
5.5 Query Federation

With use of query federation, one can distribute queries over multiple SPARQL endpoints [W3C13a]. This allows us to implement subclass reasoning, or include other data sources on an external server, without having to modify the main data store.

Query federation affects both the client- and server-side. The client needs to be aware of which portions of a query should be sent to which endpoint. As for the server-side, we may have to put up one or more additional servers to support our intended service. We will employ a basic query federation technique, where it is always known beforehand which data graph each part of the query should go to.

For an example, let us consider a modified version of the fishing sector case. This time, the main data graph only includes information about which sector each company belongs to:

```
:hansenFiskAS :hasSector :marineFishing .
:fjellfeskSA :hasSector :freshwaterFishing .
:marineFishing a :MarineFishing .
:freshwaterFishing a :FreshwaterFishing .
```

Another graph contains the subclass information:

```
:MarineFishing rdfs:subClassOf :Fishing .
:FreshwaterFishing rdfs:subClassOf :Fishing .
```

We will denote these graphs $G_1$ and $G_2$, respectively. $G_1$ and $G_2$ need not be stored on the same server. A federated query will be able to combine the two information sources, by sending separate subqueries to each endpoint:

For $G_1$:
```
?company :hasSector ?sector .
```

For $G_2$:
```
?sector a :Fishing .
```

This allows $\text{?sector}$ to be instantiated as $\text{:freshwaterFishing}$ and $\text{:marineFishing}$ from $G_2$, and $\text{?company}$ to be instantiated as $\text{:hansenFiskAS}$ and $\text{:fjellfeskSA}$ from $G_1$.

5.6 Summary

We have seen five different alternatives for supporting subclass reasoning over Linked Data, all with their own pros and cons. This
section provides a summary, highlighting the main characteristics of each approach.

**RDFS Backward Reasoning**  Server-side RDFS reasoning. Trivial to activate, and requires no further maintenance.

**RDFS Forward Reasoning**  Server-side pre-generation of all inferred triples. Potential problems with maintenance, as changes to ontology typically requires rerun of the reasoner.

**Property Paths**  Server-side SPARQL 1.1 mechanism.

**Query Rewriting**  Rewriting of queries that requires no changes to the dataset, but need for extra logic in the client.

**Query Federation**  Splitting of queries to multiple endpoints. No changes needed on the main data server, but need for additional server(s). Requires support for the SERVICE mechanism of SPARQL 1.1 on at least one of the endpoints. Still immature; varied support in current SPARQL implementations [Rak+13].
Chapter 6

Implementing Subclass Reasoning for Linked Data

This chapter presents a case study, where each of the methods for supporting subclass reasoning described in Chapter 5 are applied to the Semicolon dataset. In particular, our case will be to support subclass reasoning over the sector code class hierarchy, as described in Section 2.4. This is important in order to be able to answer queries involving the sector codes correctly.

While Chapter 5 gave a high-level description of different approaches to subclass reasoning, this chapter gives the details on how these approaches can be implemented in practice, exemplified by the sector hierarchy for companies in the Norwegian Entity Registry.

As the original Semicolon dataset does not order the sector codes into a hierarchy, we need to extend the existing dataset a bit to incorporate the subclassing logic that we want. Exactly how this extension is made is explained in the following section. Note that extending the data graph may not be needed in general, but is needed in case of the Semicolon dataset.

6.1 Modeling the NACE Hierarchy in RDF

Sectors codes are already given for each company in the existing Semicolon dataset, but they are not yet arranged in a subclass hierarchy. For instance, the following triples from the existing dataset connects the University of Oslo (which is represented by the URI unit:971035854), with the “University education” sector (NACE code 85.421):

\[
\text{unit:971035854 org:nacekode inace:85.421 .} \\
\text{inace:85.421 a nace:Nacekode .}
\]
Sector 85.421 is however not yet connected to any of the other sector codes, except that they all share nace:Nacekode as their common type.

Our proposed solution is to create an additional RDF graph where every NACE sector is represented by an RDFS class, with appropriate RDFS subclass relations between them. The sector instances, such as inace:85.421, will then receive an additional type from the newly created class hierarchy.

For instance, relevant triples for sector 85.421 that are missing in the current dataset would be:

``` yards
nace:P rdfs:subClassOf nace:Nacekode .
nace:85 rdfs:subClassOf nace:P .
nace:85.4 rdfs:subClassOf nace:85 .
nace:85.42 rdfs:subClassOf nace:85.4 .
nace:85.421 rdfs:subClassOf nace:85.42 .
inace:85.421 a nace:85.421 .
```

The new graph will need to include the above triples, plus triples constructed in a similar manner for every other NACE class in the existing dataset. Instead of listing all the triples here, the complete graph can be retrieved from [http://folk.uio.no/simenheg/nace.ttl](http://folk.uio.no/simenheg/nace.ttl) It will be referred to in subsequent sections as “the NACE graph”.

Note that this is just one way of modeling a subclass hierarchy in RDF. Other choices could be to model the hierarchy with concepts from the SKOS vocabulary[^1] or to invent and introduce new custom properties ourselves (i.e. nace:subSectorOf). We will not explore these options further, but stick to the proposed model. Even if we chose this model without exploring the alternatives, the choice is not entirely arbitrary. Advantages of this model are the mentioned formal semantics of RDFS, and its broad support in current reasoning engines. Thus our goal will not be to explore how to model subclass hierarchies in RDF, but rather to analyze how we can support subclass reasoning, given the RDF model.

The rest of this chapter presents a case study where each of the subclass reasoning approaches described in Chapter 5 are applied to support subclass reasoning over the combination of the Semicolon dataset and the NACE graph. Each of the remaining sections gives implementation details for each of the subclass reasoning approaches.

6.2 RDFS Backward Reasoning

RDFS backward reasoning is supported in several contemporary RDF databases, such as Virtuoso\(^2\), 4sr\(^3\) and Stardog\(^4\). Implementation of this approach is simple; it amounts to adding the NACE graph to the main data graph, and activating RDFS backward reasoning on the server. The server is then responsible for performing the actual inference over the newly added NACE graph.

The server will now be able to infer the triples needed to answer subclass-related queries correctly. For instance, from the NACE graph triples listed in the previous section, an RDFS backward reasoner will be able to infer the following triples about the 85.421 sector:

\[
\begin{align*}
inace:85.421 & \text{ a } nace:85.42 . \\
inace:85.421 & \text{ a } nace:85.4 . \\
inace:85.421 & \text{ a } nace:85 . \\
inace:85.421 & \text{ a } nace:P .
\end{align*}
\]

That is, the sector instance \texttt{inace:85.421} is no longer only a member of the class \texttt{nace:85.421}, but also of \texttt{nace:85.42}, \texttt{nace:85.4}, \texttt{nace:85}, and \texttt{nace:P}.

A query asking for organizations with a sector of type P ("Education"), will now correctly return organizations with sector 85.421 ("University education") as well, since that sector is now also of type P.

6.3 RDFS Forward Reasoning

This approach involves feeding the NACE graph to an RDFS forward reasoner, so that every inferred triple can be produced, and loaded into the database.

RDFS forward reasoning is supported directly in the Fuseki\(^5\) RDF server. Alternatively, an external reasoner such as Sesame\(^6\) or HermiT\(^7\) can be used to produce the inferred triples.

For the NACE graph, the inferred triples will be produced in a similar fashion to those shown in Section 6.2. We applied the HermiT reasoner to produce the inferred triples for the NACE graph. The complete inferred graph can be retrieved from \url{http://folk.uio.no/simenge/nace-inf.ttl}.
6.4 Property Paths

Utilizing property paths requires little effort once we know that the target endpoint supports them. To support subclass reasoning by property paths in the Semicolon case, all that is needed is to merge the NACE graph into the main dataset.

For the client, we have to take care during query construction. Instead of typing variables with the ordinary \texttt{rdf:type} predicate in our constructed SPARQL queries, we need allow the graph matching algorithm to follow zero or more \texttt{rdfs:subClassOf} property links, as explained in Section 5.3. As an optimization, we will only include such a path for types where subclassing is relevant.

Consider for an example a case where the user wants to retrieve every company belonging to NACE sector A. The sector variable \texttt{?X0} is permitted to be instantiate as any sector of type \texttt{nace:A}, or any subclass of \texttt{nace:A}:

\begin{verbatim}
SELE\texttt{?X WHERE { 
 ?X a org:Enhet .
 ?X org:nacekode ?X0 .
 ?X0 a/rdfs:subClassOf* nace:A .}
\end{verbatim}

6.5 Query Rewriting

We had a number of different alternatives on how to implement query rewriting, as described in Section 5.4. The following subsections treat the implementation of each of these alternatives.

Note that an optimization can be made in case of the NACE hierarchy, by observing that only the leaf nodes of the hierarchy are instantiated in the dataset. This allows us to only include sectors without any subsectors in the constructed queries. This makes the list of potential classes to consider considerably shorter, as the NACE hierarchy contains 1808 classes, but only 817 of them are leaf nodes.

For instance, the leaf nodes of the “Fishing” sector (NACE code 03.1) are “Marine fishing”, “Whaling”, and “Freshwater fishing”, with NACE codes 03.111, 03.112, and 03.120 respectively. With the suggested optimization, a query asking for organizations belonging to the fishing sector would thus amount to query for any organization, and then only keep those with a sector code which is either 03.111, 03.112, or 03.120.
6.5.1 Unions

In the case of leaf NACE classes, where no subclass reasoning is needed, we are able to retrieve an organization of a particular sector by locking the NACE code variable in the query to an instance. By allowing multiple sectors to be instantiated, we are able to emulate subclass reasoning over the NACE classes as described in Section 5.4.

In SPARQL, this effect is achieved through use of the `UNION` keyword, which allows us to specify multiple alternatives for a graph pattern [HS13, Ch. 7]. For the fishing sector example introduced in the previous section, this would mean that we allow the sector variable to be instantiated as both 03.111, 03.112, and 03.120. In SPARQL, a complete query for the fishing sector example looks like the following:

```
SELECT ?X WHERE {
  ?X a org:Enhet .
  { ?X org:nacekode inace:03.111 }
  UNION
  { ?X org:nacekode inace:03.112 }
  UNION
  { ?X org:nacekode inace:03.120 } .
  ?X0 a nace:Nacekode .
}
```

6.5.2 Instance Filtering

By using the SPARQL keyword `FILTER`, we are able to put restrictions on the solutions retrieved by a graph pattern [HS13, Ch 5.2.2]. By using the `IN` operator, we can ensure that a variable is never bound to a URI except for those specified in a URI list.

With SPARQL, it looks like the following:

```
SELECT ?X WHERE {
  ?X a org:Enhet .
  ?X org:nacekode ?X0 .
  ?X0 a nace:Nacekode .
  FILTER (?X0 IN (inace:03.111, inace:03.112, inace:03.120))
}
```

6.5.3 Attribute Filtering

As seen in Section 5.4.3, the NACE codes themselves are ordered lexicographically. This gives us a convenient way of defining the sector hierarchy implicitly through the NACE codes.
In the Semicolon dataset, the NACE code of each sector is defined in a string via the rdfs:label property. SPARQL provides operators for comparing strings lexicographically [HS13, Ch. 17.3]. With this, we are able to translate the fishing sector example from Section 5.4.3 to a working SPARQL query:

```
SELECT ?X WHERE {
  ?X a org:Enhet .
  ?X0 a nace:Nacekode .
  ?X0 rdfs:label ?code .
  FILTER (?code >= "03.1" && ?code < "03.2")
}
```

6.6 Query Federation

For this approach, we put up our own SPARQL server, serving the NACE graph through a SPARQL endpoint. We use SPARQL’s SERVICE keyword to perform federated queries across the two endpoints [W3C13a].

The client needs to be able to decide which triple patterns go to which endpoint. Two options seem appropriate for doing so:

1. Forward every query triple containing concepts that are known to have subclasses to the reasoning server.
2. Forward query triples containing hand-picked concepts to the reasoning server.

The first option requires no manual tuning, but is less flexible in the cases where the main endpoint includes subclass hierarchies that we don’t want to forward to the reasoning server. For this reason, we chose the second option for our implementation, which requires some manual tuning of the client’s data file.

Federation is achieved by means of SPARQL’s service mechanism. By using the keyword SERVICE, followed by an endpoint URI, the next graph pattern will be sent to the given endpoint [W3C13a].

Consider for instance a case where the user asks for every company with a sector belonging to NACE class A: “Agriculture, forestry and fishing”. This class spans 46 different candidate sectors, such as “Growing of rice”, “Extraction of salt”, and “Freshwater fishing”. To infer this fact, the following single triple is sent to the reasoning server:

```
?X0 a nace:A .
```

This allows $X_0$ to be instantiated as any of the subclasses of NACE class A, which can be used in further graph matching on the main server. Put together, the complete query becomes the following:
SELECT ?X WHERE {
  ?X a org:Enhet .
  ?X org:nacekode ?X0 .
  SERVICE <http://sws.ifi.uio.no/sparql/nace> {
    ?X0 a nace:A .
  }
}

6.7 Summary

This chapter covered the details of how we chose to implement each of the subclass reasoning approaches from Chapter 5 to work with the NACE class hierarchy in the Norwegian Entity Registry.

The dataset had to be augmented by an additional RDF graph, constructed by us to facilitate subclass reasoning over the NACE class hierarchy. Given the extra NACE graph, RDFS backward reasoning was trivial to support; it simply amounted to activating it in the Virtuoso server. RDFS forward reasoning required a bit more effort; we had to create an inferred NACE graph prior to loading it into the database. The property paths feature of SPARQL 1.1 worked “out of the box” with Virtuoso. With query rewriting, we had to shift the subclass reasoning logic from the server to the client, and implement the different query rewriting schemes in the client code. Query federation also required additional logic on the client side, as the client needed to decide which triples go to which endpoint. We solved this by manually marking the triples in the client’s configuration file which should be forwarded to the reasoning server.

In the next chapter, we will perform an in-depth analysis of how well each of these implementations worked in practice.
Chapter 7

Evaluation of Subclass Reasoning Approaches

In this chapter, we will evaluate all the proposed solutions to the subclass reasoning problem. The evaluation is split in two parts; we first introduce a quantitative analysis of the approaches with respect to storage space and query response time. The second part makes a more qualitative comparison of the approaches, considering for instance the perceived complexity of the approach, its flexibility, and the maintenance cost.

7.1 Performance Analysis

In this section, we aim to give an analysis of quantitative qualities of each subclass reasoning approach. We will do so by measuring different metrics in what we consider a realistic setting.

Most importantly, we will measure the query response times, a common benchmarking metric \cite[Ch. 3.3]{Jai91}. We will also consider the extra storage space required for each subclass reasoning approach.

7.1.1 Storage Space

The original Semicolon dataset contains almost 50 million triples (over 8 GB of serialized data). The NACE graph introduced in Section 6.1 which has to be added to the dataset for some of the approaches, has the modest sum of 3616 triples. RDFS forward reasoning requires the largest amount of extra storage space. The impact is however small; as seen in Table 7.1 it results in a 0.03% size increase relative to the original dataset.

The added NACE graph accounts for the 3616 extra triples in the RDFS backward reasoning, property paths, and query federation
Table 7.1: Storage space comparison

<table>
<thead>
<tr>
<th>Approach</th>
<th>Extra Triples</th>
<th>Relative Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDFS backward reasoning</td>
<td>3616</td>
<td>0.0076</td>
</tr>
<tr>
<td>RDFS forward reasoning</td>
<td>18392</td>
<td>0.0374</td>
</tr>
<tr>
<td>Property paths</td>
<td>3616</td>
<td>0.0076</td>
</tr>
<tr>
<td>Query rewriting</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Query federation</td>
<td>3616</td>
<td>0.0076</td>
</tr>
</tbody>
</table>

approaches. RDFS forward reasoning naturally requires the largest amount of extra triples, due to the inferred triples as explained in Section 7.3.2. For query rewriting, all the subclass reasoning is performed on the client side, so no extra triples have to be added to the server for this approach.

7.1.2 Query Response Time

To get an idea of how well each approach performs with regard to query response time, we performed a benchmarking experiment with the Semicolon dataset. This subsection describes the execution, and the results of this experiment.

The idea of the experiment was to set up an environment as close as possible to the one used in a real setting, and measure what impact changing the subclass reasoning approach has on the query response time. The Semicolon dataset is normally hosted on a server owned by Computas\footnote{http://data.computas.com/}, running a Virtuoso\footnote{http://virtuoso.openlinksw.com/} database to serve the RDF graph. For this reason, we decided to have a Virtuoso database in our experiment as well.

Which database system is in use is a contributing factor, but it is not our purpose here to analyze the performance of different RDF triple stores (see for instance \cite{Mir+10} for that). There are other factors to consider as well. The following list attempts to capture the most important parameters that may influence the query response time:

- Database implementation
- Server specs, like RAM, CPU speed, or operating system
- Current network load
- Current server load
- Query history
The query itself

Subclass reasoning approach

In this study, we will disregard every parameter except the query itself and the subclass reasoning approach taken, and make the simplifying assumption that the query response time only depends on the query itself and the subclass reasoning approach in use. Specifications for the server used in our experiment are listed in Table 7.2.

For the query federation approach, an additional server was used. This server ran the Joseki SPARQL server\footnote{http://www.joseki.org/} with RDFS backward reasoning activated. This server was responsible for the actual subclass reasoning, as explained in Section 6.6.

Table 7.2: Test server specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Ubuntu 12.04, with Linux 3.11.0</td>
</tr>
<tr>
<td>Database</td>
<td>Virtuoso Open Source Edition 7.1.0</td>
</tr>
<tr>
<td>Memory</td>
<td>7.7 GiB</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Core i7 (2.93 GHz × 8)</td>
</tr>
</tbody>
</table>

We could have employed an existing benchmarking suite, such as the Berlin SPARQL Benchmark\footnote{http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/} to evaluate the different subclass reasoning approaches. Though it is essential to construct a test suite that represents real world usage of the system in order to get meaningful measurements \cite[Ch. 2.2]{Jai91}. For this reason, we decided to measure only queries that are typically constructed by our Linked Open Data search tool prototype\footnote{This prototype is presented thoroughly in Chapter 8} and to create a specialized benchmarking suite based on those.

To achieve this, we used our Linked Open Data search tool prototype to answer six different questions, and collected the SPARQL queries constructed by the tool. All the questions require subclass reasoning to be answered correctly. The questions were as follows:

1. Organizations in sector “Higher education” with more than 500 employees.
2. Organizations in sector “Fishing” located in Oslo.
3. Organizations in sector “Construction” founded before year 1900.
4. Organizations in sector “Air transport” with a name containing “WING”.

\cite{Jai91}
5. Every organization in sector “Compulsory social security activities”.

6. Organizations in sector “Telecommunication”, located in Bergen, that earned more than 1,000,000 NOK in year 2010.

Question 1 involves finding organizations of 9 potential subclasses, filtered by a numerical datatype property.

Question 2 involves finding organizations of 6 potential subclasses, connected to another instance through an object property.

Question 3 involves finding organizations of 61 potential subclasses, filtered by a numerical datatype property.

Question 4 involves finding organizations of 9 potential subclasses, filtered by regular expression matches on a datatype property.

Question 5 involves finding every organization of 3 potential subclasses.

Question 6 involves finding organizations of 13 potential subclasses, connected to two other instances through object properties, one of which should be filtered on two different datatype properties.

For the raw SPARQL queries corresponding to each question, the unprocessed observation data, and the software we wrote to run the benchmarks, see: http://folk.uio.no/simenheg/subclass-reasoning-benchmark/.

For each question, and for each subclass reasoning approach, a query was constructed with our search tool prototype. Then the server was set up for each subclass reasoning approach as described in Chapter 6 before the queries were sent by turn. Query response time was measured as the elapsed time from a query was sent, until a response was received from the server. This process was repeated 100 times per question, per approach \(100 \times 6 \times 5 = 3000\) queries in total. Verification of the result was performed, but not measured as part of the response time.

While measuring the response times, we noticed a significant difference in response time between the first time a query was answered and subsequent responses to the same query. The first response to a query would typically take 2-4 times longer than the subsequent ones, probably due to caching optimizations made by the server.

To take this into consideration, we divided the experiment into two parts: measuring cold and warm response times. We define cold response time as the response time of the first time the server responds to a particular query, while a warm response occurs when the server has responded to the query before, and had an opportunity to cache it. Since both of these cases seem realistic to us in a real world scenario, we took measurements of both.
Cold Response Times

When measuring the cold response times, we wanted to model the real world scenario where a user makes a previously unseen query to the RDF database. To minimize the caching effect in the server, we restarted the server between every query sent. While doing this, we discovered that the Virtuoso database will spend 3-4 seconds extra to respond to the first couple of queries, no matter the query. To remedy this unrealistic effect that comes from restarting the server every time, we warmed up the server with five unrelated queries after each restart, before measuring the cold, uncached response time of the relevant query.

Table 7.3 lists the mean, median, minimum, maximum, and standard deviation of the cold response time observations. It is divided into six parts horizontally, so that each part represents the above questions in that order. The best times are displayed in green, while the worst times are displayed in red.
Table 7.3: Cold query response time summary for questions 1-6 (in milliseconds)

<table>
<thead>
<tr>
<th>Query</th>
<th>Approach</th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDFS backward reasoning</td>
<td>78.82</td>
<td>79.00</td>
<td>71</td>
<td>98</td>
<td>4.19</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>199.29</td>
<td>199.00</td>
<td>191</td>
<td>206</td>
<td>3.05</td>
</tr>
<tr>
<td>Q1</td>
<td>Property paths</td>
<td>292.52</td>
<td>292.00</td>
<td>285</td>
<td>306</td>
<td>3.86</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>216.17</td>
<td>216.00</td>
<td>208</td>
<td>224</td>
<td>3.42</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>246.13</td>
<td>245.00</td>
<td>237</td>
<td>318</td>
<td>8.80</td>
</tr>
<tr>
<td></td>
<td>RDFS backward reasoning</td>
<td>100.42</td>
<td>100.50</td>
<td>91</td>
<td>110</td>
<td>3.38</td>
</tr>
<tr>
<td>Q2</td>
<td>RDFS forward reasoning</td>
<td>40.45</td>
<td>41.00</td>
<td>35</td>
<td>46</td>
<td>2.52</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>395.37</td>
<td>394.00</td>
<td>389</td>
<td>416</td>
<td>5.06</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>30.63</td>
<td>31.00</td>
<td>25</td>
<td>36</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>61.49</td>
<td>60.00</td>
<td>48</td>
<td>190</td>
<td>13.59</td>
</tr>
<tr>
<td>Q3</td>
<td>RDFS backward reasoning</td>
<td>89.45</td>
<td>90.00</td>
<td>80</td>
<td>108</td>
<td>4.57</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>37.25</td>
<td>38.00</td>
<td>30</td>
<td>48</td>
<td>3.27</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>4476.68</td>
<td>4476.50</td>
<td>4242</td>
<td>4616</td>
<td>61.74</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>42.83</td>
<td>43.00</td>
<td>33</td>
<td>55</td>
<td>3.46</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>4470.99</td>
<td>4460.00</td>
<td>4371</td>
<td>5360</td>
<td>98.64</td>
</tr>
<tr>
<td>Q4</td>
<td>RDFS backward reasoning</td>
<td>55.48</td>
<td>55.50</td>
<td>49</td>
<td>66</td>
<td>3.25</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>12.53</td>
<td>12.00</td>
<td>8</td>
<td>17</td>
<td>2.52</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>5720.53</td>
<td>5708.00</td>
<td>5584</td>
<td>5932</td>
<td>80.86</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>14.72</td>
<td>15.00</td>
<td>9</td>
<td>20</td>
<td>3.11</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>5409.46</td>
<td>5399.50</td>
<td>5283</td>
<td>5698</td>
<td>70.15</td>
</tr>
<tr>
<td>Q5</td>
<td>RDFS backward reasoning</td>
<td>54.85</td>
<td>55.00</td>
<td>49</td>
<td>67</td>
<td>3.47</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>12.54</td>
<td>13.00</td>
<td>7</td>
<td>16</td>
<td>2.40</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>674.56</td>
<td>670.00</td>
<td>664</td>
<td>713</td>
<td>11.46</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>10.77</td>
<td>11.00</td>
<td>6</td>
<td>18</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>29.34</td>
<td>29.00</td>
<td>21</td>
<td>90</td>
<td>7.50</td>
</tr>
<tr>
<td>Q6</td>
<td>RDFS backward reasoning</td>
<td>739.98</td>
<td>746.00</td>
<td>541</td>
<td>861</td>
<td>77.04</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>938.89</td>
<td>939.00</td>
<td>900</td>
<td>966</td>
<td>9.33</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>2365.15</td>
<td>2360.50</td>
<td>2324</td>
<td>2433</td>
<td>21.10</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>1028.59</td>
<td>1027.50</td>
<td>983</td>
<td>1071</td>
<td>13.11</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>988.74</td>
<td>981.00</td>
<td>952</td>
<td>1164</td>
<td>32.40</td>
</tr>
</tbody>
</table>
Warm Response Times

When measuring the warm response times, we wanted to model the real world scenario where a user makes a query that has previously been responded to by the database. This was done by warming up the server with all the six predefined queries, before measuring the response time of the current relevant query.

Table 7.4 lists the mean, median, minimum, maximum, and standard deviation of the warm response time observations. Like for the cold response time table, it is divided into six parts horizontally, so that each part represents the above questions in that order. The best times are displayed in green, while the worst times are displayed in red.
Table 7.4: Warm query response time summary for questions 1-6 (in milliseconds)

<table>
<thead>
<tr>
<th>Query</th>
<th>Approach</th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>RDFS backward reasoning</td>
<td>42.66</td>
<td>43.00</td>
<td>38</td>
<td>46</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>11.53</td>
<td>11.00</td>
<td>6</td>
<td>17</td>
<td>2.87</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>45.88</td>
<td>44.00</td>
<td>43</td>
<td>68</td>
<td>3.66</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>15.81</td>
<td>15.00</td>
<td>12</td>
<td>39</td>
<td>2.95</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>94.48</td>
<td>58.00</td>
<td>36</td>
<td>508</td>
<td>96.18</td>
</tr>
<tr>
<td>Q2</td>
<td>RDFS backward reasoning</td>
<td>49.89</td>
<td>49.00</td>
<td>46</td>
<td>53</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>20.56</td>
<td>20.00</td>
<td>19</td>
<td>26</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>344.17</td>
<td>344.00</td>
<td>340</td>
<td>355</td>
<td>2.16</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>17.23</td>
<td>17.50</td>
<td>15</td>
<td>19</td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>102.20</td>
<td>51.50</td>
<td>35</td>
<td>3352</td>
<td>354.24</td>
</tr>
<tr>
<td>Q3</td>
<td>RDFS backward reasoning</td>
<td>56.74</td>
<td>57.00</td>
<td>54</td>
<td>63</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>23.86</td>
<td>24.00</td>
<td>22</td>
<td>29</td>
<td>1.31</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>4468.77</td>
<td>4469.00</td>
<td>4429</td>
<td>4506</td>
<td>13.83</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>28.06</td>
<td>29.00</td>
<td>24</td>
<td>33</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>4836.00</td>
<td>4480.00</td>
<td>4424</td>
<td>17358</td>
<td>1639.60</td>
</tr>
<tr>
<td>Q4</td>
<td>RDFS backward reasoning</td>
<td>29.29</td>
<td>30.00</td>
<td>26</td>
<td>32</td>
<td>1.63</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>8.27</td>
<td>9.00</td>
<td>4</td>
<td>33</td>
<td>3.38</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>5921.08</td>
<td>5917.00</td>
<td>5903</td>
<td>6049</td>
<td>17.51</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>8.20</td>
<td>7.00</td>
<td>5</td>
<td>13</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>5622.61</td>
<td>5599.50</td>
<td>5581</td>
<td>6932</td>
<td>136.17</td>
</tr>
<tr>
<td>Q5</td>
<td>RDFS backward reasoning</td>
<td>28.93</td>
<td>28.00</td>
<td>27</td>
<td>35</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>8.54</td>
<td>9.00</td>
<td>4</td>
<td>11</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>663.24</td>
<td>663.00</td>
<td>660</td>
<td>675</td>
<td>2.62</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>8.46</td>
<td>9.00</td>
<td>4</td>
<td>11</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>29.92</td>
<td>27.00</td>
<td>18</td>
<td>237</td>
<td>21.64</td>
</tr>
<tr>
<td>Q6</td>
<td>RDFS backward reasoning</td>
<td>304.51</td>
<td>305.00</td>
<td>298</td>
<td>314</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>RDFS forward reasoning</td>
<td>351.59</td>
<td>352.00</td>
<td>346</td>
<td>361</td>
<td>3.17</td>
</tr>
<tr>
<td></td>
<td>Property paths</td>
<td>1701.29</td>
<td>1700.00</td>
<td>1696</td>
<td>1719</td>
<td>3.68</td>
</tr>
<tr>
<td></td>
<td>Query rewriting</td>
<td>291.78</td>
<td>292.00</td>
<td>287</td>
<td>296</td>
<td>2.78</td>
</tr>
<tr>
<td></td>
<td>Query federation</td>
<td>334.42</td>
<td>330.00</td>
<td>314</td>
<td>428</td>
<td>16.68</td>
</tr>
</tbody>
</table>
Observations

Here follows some observations based on the query response times measured for each question.

**Question 1** Every approach had a cold response time below 1/3 seconds on average. For the warm responses, every approach had an average response time below 1/10 of a second. In the cold response test, backward reasoning provided the best response times, while forward reasoning provided the best response times in the warm response test.

The warm response time measurements for the query federation approach reveals something interesting. Once in a while, a query would take more than 5 times longer than the average for the same query, and more than 10 times longer than the lowest measured response time. This gives the query federation approach the highest variability in this test, with a standard deviation of almost 1/10 of a second.

**Question 2** Property paths performed worse than the others on average with respect to both warm and cold response time, with a mean response time of over 1/3 of a second in both cases. Query rewriting provided the fastest response times any case.

Again we observe very high variability in the warm response times of the federated queries.

**Question 3** The increased amount of potential subclasses in this query seem to affect the property paths- and query federation approaches badly, both having a mean warm- and cold response times of over 4 seconds. All the other approaches spent below 1/10 of a second on average, with forward reasoning performing best in every respect.

The query federation approach was again the most unstable with the respect to warm query response time, taking between 4 and 17.5 seconds to respond.

**Question 4** Again, both the property paths- and the query federation approach performed considerably worse than the others, with a mean warm- and cold response time of over 5 seconds. Forward reasoning and query rewriting provided the best response times.

**Question 5** The property paths approach performed worst, with average response times of more than half a second. Again, forward reasoning and query rewriting provided the best response times, with warm response times as good as 4/1000 of a second.

**Question 6** For this question, property paths performed considerably worse than the others. Every approach except property paths had a mean cold response time of around a second, while property
paths spent over 2 seconds on average. Every approach except property paths had a mean warm response time of around 1/3 of a second, while property paths spent over 1.5 seconds on average.

Table 7.5: Total cold response time means

<table>
<thead>
<tr>
<th>Approach</th>
<th>Mean of Means</th>
<th>SD mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward reasoning</td>
<td>186.50 ms</td>
<td>15.98 ms</td>
</tr>
<tr>
<td>Forward reasoning</td>
<td>206.83 ms</td>
<td>3.85 ms</td>
</tr>
<tr>
<td>Property paths</td>
<td>2320.80 ms</td>
<td>30.68 ms</td>
</tr>
<tr>
<td>Query rewriting</td>
<td>223.95 ms</td>
<td>4.80 ms</td>
</tr>
<tr>
<td>Query federation</td>
<td>1867.69 ms</td>
<td>38.51 ms</td>
</tr>
</tbody>
</table>

Table 7.5 lists the mean of mean cold response times, and the mean of the standard deviations for each approach.

These results show a clear performance difference between the approaches that spend around 1/5 of a second per query on average, versus those that spend over 1.5 seconds on average. Forward reasoning, backward reasoning, and query rewriting belong to the former group, while property paths and query federation belong to the latter. Among all, forward reasoning and query rewriting had the lowest standard deviations on average, meaning they provided the most stable response times. Query federation provided the least stable response times. This was expected, since query federation relies on the cooperation of two SPARQL databases instead of just one.

Table 7.6: Total warm response time means in comparison to the cold response time means

<table>
<thead>
<tr>
<th>Approach</th>
<th>Mean of Means</th>
<th>Ratio to Cold Means</th>
<th>SD mean</th>
<th>Ratio to Cold SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward reasoning</td>
<td>85.34 ms</td>
<td>0.46</td>
<td>1.98 ms</td>
<td>0.12</td>
</tr>
<tr>
<td>Forward reasoning</td>
<td>70.73 ms</td>
<td>0.34</td>
<td>2.33 ms</td>
<td>0.61</td>
</tr>
<tr>
<td>Property paths</td>
<td>2190.74 ms</td>
<td>0.94</td>
<td>7.24 ms</td>
<td>0.24</td>
</tr>
<tr>
<td>Query rewriting</td>
<td>61.59 ms</td>
<td>0.28</td>
<td>2.14 ms</td>
<td>0.45</td>
</tr>
<tr>
<td>Query federation</td>
<td>1836.61 ms</td>
<td>0.98</td>
<td>377.51 ms</td>
<td>9.80</td>
</tr>
</tbody>
</table>

Table 7.6 lists the mean of mean warm response times, and the mean of the standard deviations for each approach.

The third column lists the ratio of the warm response time mean of means to the cold response time mean of means. From this, we see that every approach has improved over the cold response time,
but the property paths and query federation approaches improved only marginally. Query rewriting had the largest performance gain, spending only 28% of the time on average when the server was warmed up.

The fifth column lists the ratio of the warm response time mean of standard deviations to the cold response time mean of standard deviations. Curiously, every approach got more stable query response times except query federation, which got almost 10 times larger standard deviations on average.

To sum up, we would regard the response time- and standard deviation averages of backward reasoning, forward reasoning and query rewriting as adequate for our purpose. Property paths and query federation provided worse response times than all of the above on average, with query federation having the additional downside of providing the least stable response times.

### 7.2 Qualitative Comparison

Table 7.7: Qualitative assessment of subclass reasoning approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Conceptual Complexity</th>
<th>Setup Effort</th>
<th>Flexibility</th>
<th>Maint. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward reasoning</td>
<td>Low</td>
<td>Very low</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td>Forward reasoning</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Property paths</td>
<td>Low</td>
<td>None</td>
<td>Very high</td>
<td>None</td>
</tr>
<tr>
<td>Query rewriting</td>
<td>Medium</td>
<td>Medium</td>
<td>Very high</td>
<td>Medium</td>
</tr>
<tr>
<td>Query federation</td>
<td>High</td>
<td>High</td>
<td>Very high</td>
<td>None</td>
</tr>
</tbody>
</table>

In this section, we will compare our experience with each of the subclass reasoning approaches with respect to qualitative measures, such as their perceived complexity, setup effort, generality, flexibility, and maintenance cost. Our assessments are summarized in Table 7.7.

**RDFS backward reasoning** support is built into the Virtuoso database system, which made it easy for us to apply it to the NACE graph. This approach required no effort apart from activating RDFS backward reasoning on the server. It requires no maintenance after this point, as changes to the dataset are taken into consideration automatically by the server.

**RDFS forward reasoning** was performed by the external HermiT reasoner in our experiment, as we could not find any support for forward reasoning directly in Virtuoso. Adding the inferred graph to the database was a small manual task. While the approach is conceptually simple, we would have to watch out for changes in the ontology, which
may have us rerun the reasoner on occasion to keep the inferred graph updated. This can be problematic [Heb+09, p. 150].

**Property paths** was perceived as the simplest of all the approaches. Once we configured the search tool to include property paths in the constructed queries, this approach simply worked without any further configuration on the server. It is extensible in that we are not limited to creating property paths of just RDFS “subclass of” relations. For instance, we could easily extend this approach to emulate reasoning for the “broader” relation from the SKOS vocabulary as well.

**Instance filtering by query rewriting** was perceived as conceptually complex. It requires the client to maintain a complete list of every possible instance of each class in the dataset. While this list can be constructed automatically, it has to be redone every time an individual or a class is added – or removed – from the dataset. The approach proved very flexible, in that we are free to define subclass hierarchies for instances independently of existing- or non-existing hierarchies on the server. For this reason, we went with this approach during the user study presented in Chapter 9, where we were missing access to the main Semicolon data server.

**Query federation** turned out to give rise to several problems. First off, not every SPARQL implementation supports query federation yet, including the main Semicolon data server. We discovered that the performance of our federated queries were heavily dependent on the order of the **SERVICE**-clauses, providing unsatisfactory response times for certain types of queries. We also encountered compatibility issues between different SPARQL implementations. Introducing an additional server dependency adds a new link to the chain of potential failures. On the bright side, query federation can provide very high flexibility when wanting to augment an existing data store, without necessarily having access to change the data store directly.

### 7.3 Summary

This section gives a summary of the experiences we made with each approach to support subclass reasoning for Linked Data. Based on our quantitative and qualitative assessments, we will give our recommendations to when (if ever) an approach is suitable.

#### 7.3.1 RDFS Backward Reasoning

RDFS backward reasoning is broadly supported in contemporary RDF databases, and is very simple to set up. In the Semicolon test...
backward reasoning provided good query response times. When possible, this would be our recommended approach, due to its simplicity and decent response times.

### 7.3.2 RDFS Forward Reasoning

RDFS forward reasoning does not seem to be as broadly supported as RDFS backward reasoning in contemporary RDF databases. One RDF database provider, Franz Inc., states the maintenance problem of forward reasoning mentioned in Section 7.3.2 as their prime reason for not supporting forward reasoning in their AllegroGraph RDF database [Inc14].

Although it proved to be a negligible factor in the Semicolon test case, the additional required storage space must also be considered. If the inferred RDF graph becomes too big, it might cancel out the performance gain of choosing forward reasoning over backward reasoning [Heb+09, p. 150].

In the Semicolon test case, RDFS forward reasoning provided both faster, and more stable response times than RDFS backward reasoning. Despite this, we would still recommend RDFS backward reasoning over forward reasoning in general, because of the mentioned impracticalities.

RDFS forward reasoning might still be preferable when the dataset is known to be unchanging over large periods of time. In such cases, we imagine that the performance gain can out-weight the additional maintenance cost in the long run.

### 7.3.3 Property Paths

While we deem the property path approach as an elegant way of emulating subclass reasoning in RDF databases that don’t support reasoning directly, it performed worst in the Semicolon test case with respect to query response time. With an average response time of over two seconds, we considered this approach unsuitable for the Semicolon case.

That being said, our tests only considered the Virtuoso RDF database. If another database implementation provides a more efficient property path implementation, it might again prove to be a viable option for supporting subclass reasoning.

### 7.3.4 Query Rewriting

Instance filtering by query rewriting does require a full analysis of the dataset to predetermine all the available subclass hierarchies and their
instances. The analysis must be redone whenever changes are made to the subclass hierarchies, or their instances.

This approach was attractive to us in the Semicolon case because the work is done entirely on the client side, and we didn't have access to the main data server. Additionally, it showed to give very fast and stable query response times. Based on these factors, instance filtering by query rewriting would be our recommended approach for supporting subclass reasoning when the data store cannot be modified.

### 7.3.5 Query Federation

Query federation requires the availability of an additional server, and setting it up to support subclass reasoning, for instance by RDFS backward- or forward reasoning. Similarly to query rewriting, we consider the biggest strength of query federation to be the independence from the main data server.

But as we elaborated in Section 7.2, this approach gave rise to too many problems in the Semicolon case to be practical. As with the property path approach, the query response times and response time stability were also inadequate in our case study. Until the federation technology matures, we would not recommend using query federation to support subclass reasoning.
Chapter 8

Linked Open Data Search Tool Prototype

As we saw in Chapter 1, there is a large (and growing!) amount of data being published as Linked Open Data, yet it is still inaccessible to the general public due to the lack of suitable user interfaces. The LogID group is researching on how to make graphical user interfaces to enable end-users to explore and exploit Linked Open Data sources, such as the Norwegian Entity Registry, without requiring knowledge of formal query languages.

Several such interfaces already exist, but as we see it, each existing approach has its shortcomings. Figure 8.1 plots some of the approaches that can be taken in order to support query formulation for Linked Data, arranged by their perceived usability and expressivity.

In Section 8.1, we review these approaches, give examples of existing interfaces for each approach, and explain their shortcomings, before we give a detailed overview of our chosen approach in Section 8.2.

In Section 8.3, we give an overview of the architecture of our search tool prototypes. The tools and their user interfaces are configured in a semi-automatic manner by means of a SPARQL endpoint analyzing component, developed as a part of the work for this thesis. It is intended to be general enough to adapt to any given Linked Open Data store with a SPARQL endpoint. Technical details about the SPARQL endpoint analyzer are given in Section 8.3.1.

In Section 8.4, we show that our search tool prototypes can be adapted to work with Linked Open Data sets other than the Semicolon one.

Lastly, Section 8.5 describes how we chose to implement support for subclass hierarchies in the user interface of the two prototypes; an important step on the path to support subclass reasoning.
The ultimate goal of this chapter is to have search tool prototypes capable of searching – and performing subclass reasoning – over arbitrary Linked Open Data sets.

8.1 Existing Approaches

![Diagram of Usability and Expressivity](image)

**Figure 8.1: Usability and expressivity of approaches for querying Linked Data (derived from table in [Veg+14])**

**Keyword search** lets the user specify a list of words, not necessarily resembling natural language sentences, which are used in trying to collect the most relevant answers for the user. This approach to information retrieval has been popularized by Web search engines such as Google [Hea09, Ch. 4.1.2], so we can expect users to already be familiar with the concept. The usability is increased by hiding the underlying data structure; users don’t have to be aware of how the data is represented when entering a list of words. In the case of querying Linked Data, keyword search approaches drastically increase the usability compared to querying by using formal query languages, but at the cost of lowering the expressivity. But even at a lower expressivity rate, experiments like the one presented in [She+11] shows that keyword-based searches can yield acceptable results when querying Linked Data.

**Faceted search** lets the user search datasets by putting restrictions on predefined categories called *facets* [Hea09, Ch. 8.6]. This
approach is often applied for product exploration in online shopping sites on the Web, such as Amazon[^2] or the Norwegian FINN.no[^3] web site. For instance, when searching for books at an online store, relevant facets could be price range, author, book format, text language and whether international shipping is available. By this example, we see that facet restriction can both be number ranges (e.g. price), concept refinements (e.g. book format), or Boolean selections (e.g. international shipping). While faceted search might be more expressive and fine-grained than keyword searches for searching Linked Data, it cannot match the full expressivity of SPARQL. This is partly because faceted search works with refinement of single concepts, while in Linked Data queries we often want to link different concepts together. The Virtuoso Facets Web Service[^4] provides an interface for faceted searching of Linked Data.

**Natural language interfaces (NLIs)** allow users to search datasets by formulating their queries in natural language. Like keyword searches, it hides the complexity of the underlying data structure from the users, and provides instead a natural and intuitive interface for querying the data. Natural languages are however inherently ambiguous, meaning that we have to trade away exactness of the constructed queries for the increased usability. The interfaces can be made more accurate by elaborating their design, for instance by having users clarify their queries whenever ambiguity arises. Developing accurate natural languages interfaces is however a very complex and time-consuming task [^KB07].

**Special purpose user interfaces** denotes interfaces tailor-made for a target domain. Well crafted special purpose user interfaces have the potential for high expressivity, while retaining the usability. On the flip side, they typically require a large development effort, and they can not be adapted to new domains easily.

**Visual SPARQL editors** provide high expressivity by allowing direct manipulation of SPARQL queries through a visual user interface. iSPARQL[^5] is one such interface. This approach may increase the usability, but still requires users to have at least some understanding of the underlying RDF/SPARQL mechanisms, which might make this approach unfeasible in trying to expose Linked Data sources to the general public.

**SPARQL editors** provide users with the full power of SPARQL, by serving as a tool to edit raw SPARQL queries. Indeed, any raw text editor might serve as a SPARQL editor, but they may also
include features such as syntax highlighting or context-aware text completion to increase the usability. Either way, SPARQL editors require SPARQL skills way beyond what we can expect from the average citizen.

8.2 Our Approach

The LogID group is researching on how to make graphical user interfaces to enable end-users to explore and exploit Linked Open Data sources, such as the Norwegian Entity Registry, without requiring knowledge of formal query languages.

Two prototypes have been developed as part of this effort. Active development of both of these tools has been part of the work for this thesis. The first search tool prototype is PepeSearch6: a graphical web application geared toward the “man on the street” [Veg+14]. The second prototype is a more advanced version, targeting “ambitious” users. It bears resemblance to the approach taken in the Optique research project [Soy+13b].

As discussed in Chapter 1, formal query languages like SPARQL are indeed very expressive, but we can not expect the “man on the street” to know them. Thus the Linked Open Data stores that are only accessible through SPARQL become virtually useless to the general public. Keyword-based search interfaces, faceted search, or natural language approaches certainly lower the complexity, but at the expense of the expressivity. Visual SPARQL editors preserve much of the expressivity of SPARQL, but the usability remains too low.

Our prototypes try to strike a balance between usability and expressivity. This is illustrated in Figure 8.1 where our prototypes try to match the expressivity and usability of special purpose interfaces, while trying to minimize the manual work needed to adapt it to new datasets. That is, we try to reach the upper-right corner of the “Special purpose interfaces” rectangle in the figure.

The prototypes draw inspiration from faceted search mechanisms, while aiming to be more expressive than existing faceted-based search approaches for Linked Open Data. Unlike pure faceted search tools, our tools allow selection of multiple linked classes, each filtered by their own facets.

Two variants of the search tool prototype have been developed, as mentioned in the beginning of this section. Section 8.2.1 presents PepeSearch; a Linked Open Data search tool for the “man on the street”.

---

6Pepe is the pet form of José, a common Spanish name; PepeSearch is a search tool for “the common man”
while section 8.2.2 presents the advanced version of our search tool prototype.

### 8.2.1 PepeSearch

![Image](image_placeholder)

**Figure 8.2:** Using PepeSearch to search for organizations in the Semicolon dataset

PepeSearch is the simpler of the two prototypes, aiming for a low entry threshold for casual users. The main purpose of PepeSearch is to ease the access to Linked Open Data stores for the general public. An example of PepeSearch in use is illustrated in Figure 8.2 where the tool has been configured to search organizations in the Semicolon dataset. This facet page was generated automatically because the class “Organization” was found in the dataset. The attributes “name”, “organization number”, and “date of founding” are among the datatype properties belonging to the class. Not also that the numerical limits for the “date of founding” facet correspond to the minimum and maximum values found in the dataset. “Sector”, “Organizational form”, and so on, are classes related to “Organization” by object properties.

If the “Get results!” button is clicked without any refinements to the attributes, the user will retrieve a list of every organization found in the Semicolon dataset (this is not entirely true, since the tool will put a hard limit on the total number of results to retrieve). Optionally, she may refine the results by putting restrictions on the attributes. For instance, if the “date of founding” attribute is adjusted, only organizations founded within the given timespan is retrieved.

When the “Get results!” button is pressed, PepeSearch will create
Figure 8.3: Results after querying for organizations founded between year 1790 and 1800

A SPARQL query corresponding to the facets and attribute restrictions set by the user. Next, the SPARQL query is sent to a server, where the results are collected, and sent back to PepeSearch. The results are then presented to the user in a table, like the one shown in Figure 8.3. Note that the concepts are links; they can be clicked for further exploration of their associated properties.

A live demo of PepeSearch configured for searching the Semicolon dataset is available at [http://sws.ifi.uio.no/project/semicolon/search/](http://sws.ifi.uio.no/project/semicolon/search/). The complete source code of PepeSearch is published at GitHub under a free software license.

### 8.2.2 Advanced Search Tool

The advanced version of the search tool prototype works much in the same way as PepeSearch, but in addition it displays a figure representing the current SPARQL graph during query construction, which can be manipulated by the user. Concepts are shown as nodes, while object properties are displayed as edges between them. Datatype property restrictions on concepts are drawn inside their respective nodes. Figure 8.4 shows the advanced prototype in use. The shown query will retrieve every organization in the higher education sector that was founded between year 1753 and 1900, and is located in Oslo.

While PepeSearch targets casual users, this version is intended to appeal to users with more ambitious goals, such as students, researchers, or journalists. Its main purpose is to serve as a tool for search and exploration of any Linked Open Data store, intuitive to people with previous experience in Semantic Web technologies. It

is more expressive in that it can link together concepts in ways not possible with PepeSearch. By the increased expressivity, we expect the complexity and learning curve to increase accordingly.

For the advanced users, our hope is that the benefits of having a visual representation of the query out-weight the increased complexity. The query graph closely resembles the underlying RDF structure, which we hope will appeal to those with previous RDF/Semantic Web experience.

A live demo of the advanced search tool configured for searching the Semicolon dataset is available at [http://sws.ifi.uio.no/project/semicolon/advancedsearch/](http://sws.ifi.uio.no/project/semicolon/advancedsearch/).

### 8.3 Prototype Architecture

Both search tool prototypes are programmed in the JavaScript programming language. The jQuery Mobile framework is employed to make a user interface that is portable across desktop computers, smartphones and tablets. A strong motivator for choosing these frameworks was that we could reuse code from a previous prototype developed as part of the Optique project (see [Soy+13b]). Additional JavaScript libraries employed include mustache.js for HTML template support, and Underscore for functional programming support in JavaScript. To facilitate completion in text fields, we employ the Apache Solr platform.

In order to adapt to different datasets, the search tools need a general
method of analyzing SPARQL endpoints, and extracting relevant information from them. This is the role of the SPARQL endpoint analyzer component, shown at the top of Figure 8.5. Development of this component has been a large part of the work for this thesis. Details about the workings of this component are given in the following section.

8.3.1 SPARQL Endpoint Analyzer

While the search tools themselves are written in JavaScript, the SPARQL endpoint analyzer is an independent component, and is written in the Common Lisp\(^\text{12}\) programming language. Technical documentation of the tool is given in Appendix B: SPARQL Endpoint Analyzer Manual. The full source code is available at GitHub\(^\text{13}\) under a free software license.

The SPARQL endpoint analyzer can be configured to retrieve information from any given SPARQL endpoint. As mentioned in the previous section, the intention is to retrieve information from the endpoint that is relevant for configuring the search tool user interfaces. Data from the SPARQL endpoint is retrieved – and used to configure the search tools – by the following strategy:

**Classes** are mapped to user interface (UI) classes.

**Subclasses** of every class are mapped to UI class facets. Section 7.2 elaborates on how subclass hierarchies are presented in the interface.

---

12 [http://common-lisp.net/](http://common-lisp.net/)
13 [https://github.com/simenheg/sparql-endpoint-analyzer](https://github.com/simenheg/sparql-endpoint-analyzer)
Datatype properties of every class are mapped to UI class facets. Literal types are retrieved as well, in order to support different displaying schemes, such as text-fields for strings, or slider bars for numerals. When the literal type is numeric, the minimum and maximum values of that property are recorded to adjust the slider bar accordingly.

Object properties between classes are used to link the corresponding UI classes together. PepeSearch does not distinguish between incoming- and outgoing links, while the advanced search tool does. With “incoming links”, we denote properties of which the class is in object position, while “outgoing links” denote properties of which the class is in subject position.

All this information is retrieved by the SPARQL endpoint analyzer, and made available to the search tools in the standardized JavaScript Object Notation (JSON)[14] format. For instance, the JSON-entry for the “Organization” class found in the Semicolon dataset looks like the following:

```
{
    "id": "org_Enhet",
    "uri": "http://data.computas.com/informasjonsmodell/organisasjon/Enhet",
    "label": {"en": "Organization", "nb": "Enhet"},
    "display": "org_navn",
    "primary": true
}
```

The shown fields all play a role in configuring how this class is displayed in the user interface. Similar entries are made for every other class found in the dataset. We will not dive further into the details of this data format here. See [http://folk.uio.no/simenheg/semicolon.js](http://folk.uio.no/simenheg/semicolon.js) for a documented example of the JSON file used to configure PepeSearch for the Semicolon dataset.

In the future, we would like to change our data format to a standardized format instead of our proprietary one. JSON-LD[15] a newly proposed format for serializing Linked Data, seems like a good candidate.

8.4 Adapting to Other Linked Open Data Stores

While our main case study has been to support query formulation and subclass reasoning for the Semicolon dataset, we intended our prototypes to be adaptable to arbitrary Linked Open Data stores. To

put our tools to the test, we tried to employ them with two additional endpoints: the Semantic Web Dog Food Corpus and a Linked Open Data version of the FactPages from the Norwegian Petroleum Directorate.

The Semantic Web Dog Food Corpus provides a Linked Open Data source for information about papers, conferences, persons, and workshops in the Semantic Web research field. In Figure 8.6, the advanced search tool has been successfully configured to search the Semantic Web Dog Food Corpus. The shown query will result in a list of all registered persons and papers where the person is the author of a paper that has the word “SPARQL” in its title.

The FactPages from the Norwegian Petroleum Directorate contain public information regarding petroleum activities in Norway. Among the concepts recorded here are oil fields, involved companies, wells, and wellbores. Configuring the search tools for the FactPages with the SPARQL endpoint analyzer was successful, but with over 170 classes, over 50 possible links between some of them, and equally many datatype properties, the interface approaches the brink of what it can handle. For the heaviest classes, the interface spends tens of seconds to load. Figure 8.7 shows some of the many attributes that the class of “Wellbore oil samples” can be refined by.

Trying to adapt our prototypes to different endpoints was a fruitful exercise. It revealed portability issues in the SPARQL endpoint analyzer component, which were later fixed. We hope that by continuing

---

16 http://data.semanticweb.org/
17 http://factpages.npd.no/
18 http://www.npd.no/
to test the tools with different endpoints, we will be able to increase the robustness and adaptability of the tools.

8.5 Subclass User Interface

To extend the PepeSearch prototype to support subclass reasoning, some new interface components had to be developed. We chose to present the subclass hierarchy in a foldable menu. Similar hierarchical subclass interfaces are found, for instance, in the Protégé\textsuperscript{19} ontology editor, and in the Eclipse\textsuperscript{20} integrated development environment.

Figure 8.8 shows the interface in use, where a user is refining the sector class. Upon expanding the “Education” sector, the relevant subclasses are displayed and selectable (“Other education”, “Pre-primary education”, and so on).

It is also possible to search the subclass hierarchy through a free-text search bar. When the user enters a search string there, the interface will show a flat list of every class with a name matching the given string.

Note that this interface supports only single-class selection. While multi-class selection may be a desirable feature, we had to consider the trade-off of increasing the user interface complexity.

\textsuperscript{19}http://protege.stanford.edu/
\textsuperscript{20}http://www.eclipse.org/
8.6 Summary

We have presented two Linked Open Data search tool prototypes. One is geared toward casual users, while the other targets more ambitious use cases. We showed how the tools enabled users to search the Norwegian Entity Registry through a visual query interface, and that the tools were able to adapt to other Linked Open Data sets, such as the Semantic Web Dog Food Corpus and the FactPages from the Norwegian Petroleum Directorate. We also showed how support for subclass hierarchies was integrated in the interface.

In the next chapter, we will present the results from a user study of the two tools. The usability of the subclass interface as a whole was among the questions that we addressed in this study.
Chapter 9

Evaluation of Subclass Interface

A user study was conducted by the LogID group to assess the usability of our search tool prototypes [Veg14]. 15 persons volunteered to participate in the study. The participants were asked to use the tools to attempt to answer six different questions, crafted by us in attempt to cover as broad a range of use cases as possible. The exact phrasings of the questions were as follows:

1. Which wireless telecommunications companies settled in Oslo have above 40 employees?
2. Which companies does “Svein Rennemo” lead?
3. Which Norwegian companies had a net income of more than 1,000,000,000 (one billion) Norwegian kroner in 2008, but no more than 4 employees?
4. Which are the eight oldest accommodation establishments (hotels or other) in Tromsø?
5. Which companies are settled in Brønnøy, street Industriveien 30?
6. Which were the seven manufacturing companies of dairy and ice-cream products with the highest net income in 2010?

The details of the experimental setup are given in Section 9.1. For questions 1, 4, and 6, we expected the query formulation process to become significantly easier if the user took use of the subclass interface. That is, the sectors “Manufacture of dairy and ice-cream”, “Wireless telecommunication activities” and “Accommodation” can be selected through the subclass user interface.

Whether the users actually perceived the subclass interface as useful is examined in Sections 9.2 and 9.3. The measured effectiveness of using the subclass interface to formulate queries is presented in Section 9.4.
9.1 Experimental Setup

As subclass hierarchies were relevant to query formulation in three of the six given questions, it was essential to have a working subclass reasoning approach during the user study. We initially set up a federated query solution, but we then ran into character encoding issues due to having different SPARQL implementations on each server. These issues were resolved by switching to instance filtering by query rewriting, as described in Section 5.4.2.

Both PepeSearch and the advanced version of the search tool were used during the test. The participants received no prior training with the tools, except a visual tutorial explaining the different components of the user interface. Every participant had to answer all the six tasks, but the order of the tasks, and the assigned version of the tool was alternated. Each participant would use one version of the tool for the three first tasks, and the other tool for the three remaining tasks.

Table 9.1: Task orderings forming a Latin square

<table>
<thead>
<tr>
<th>ID</th>
<th>Task order</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>P2</td>
<td>2 3 4 5 6 1</td>
</tr>
<tr>
<td>P3</td>
<td>3 4 5 6 1 2</td>
</tr>
<tr>
<td>P4</td>
<td>4 5 6 1 2 3</td>
</tr>
<tr>
<td>P5</td>
<td>5 6 1 2 3 4</td>
</tr>
<tr>
<td>P6</td>
<td>6 1 2 3 4 5</td>
</tr>
</tbody>
</table>

To randomize the effect that the task order has on the performance, the task order was picked to form a Latin square, as shown in Table 9.1. A Latin square is an $n \times n$ table of $n$ different items where each item is found once in every row and once in every column [Wei06]. After assigning the six orderings from the Latin square to the six first participants, the starting tool was switched, and the Latin square orders repeated. This ensures that bias from the task- and tool order is spread out evenly.

9.2 User Evaluation of PepeSearch

The users were asked a set of questions specifically about each search tool after completing three tasks, and after completing all six tasks. We will not elaborate on the results from those questions here, as they are not directly relevant for this thesis (see [Veg14] for that). Two questions gave rise to answers relevant to the subclass interface however. We will
present those results here, before elaborating on the questions directly targeting the subclass interface in Section 9.3.

Table 9.2: Selected free-text answers to the question “What did you dislike more about this tool [PepeSearch]?”

<table>
<thead>
<tr>
<th>ID</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>It was difficult to be very specific, you would get no results. I also did not understand how to search on several &quot;sektor&quot; at once - something I think could be very useful.</td>
</tr>
<tr>
<td>P19</td>
<td>I had troubles with the accommodation query. I wanted companies that were either in the &quot;Overnattningsvirksomhet&quot; sector, or the &quot;Hotellvirksomhet&quot; sector. I had to manually union the results of two queries with the different sectors. I couldn't figure out how to do this in the tool (maybe you can't?).</td>
</tr>
<tr>
<td>P23</td>
<td>You could have more than one of the same category. For example, you could not choose hotel as one category and then camping in the same query.</td>
</tr>
</tbody>
</table>

Table 9.3: Selected free-text answers to the question “How would you improve this tool [PepeSearch]?”

<table>
<thead>
<tr>
<th>ID</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>P19</td>
<td>Add the possibility to either union queries, or better, make you able to say for example &quot;sector is 'Overnattningsvirksomhet' OR sector is 'Hotellvirksomhet'&quot;. This may come down to query complexity though?</td>
</tr>
</tbody>
</table>

The responses listed in Table 9.2 and Table 9.3 show that some users had a fundamental misconception of how the sector hierarchy works. For instance, participant P19 complains that he had to merge the results of searching for organizations in both the “Accommodation” sector and the “Hotels and similar accommodation” sector, without realizing that organizations in the “Hotels and similar accommodation” are automatically members of the “Accommodation” sector as well. In other words, it seems that these users were unaware of how the subclass hierarchy works, or unaware that it exists at all. This indicates that the PepeSearch interface fails to communicate how subclass reasoning works to some users.

Participant P19 also touches upon the interesting idea of supporting unions or multiple class selection during query formulation. That is, to specify more than one alternative for a facet, or even for a link. The current prototypes work by conjunction, so every time the user adds another facet restriction, the list of results narrows. We decided early
on not to support unions in our query formulation tools, as it would add a whole new layer of complexity.

9.3 User Evaluation of Subclass Interface

After submitting all their answers to the six tasks, the users were asked some questions addressing the user experience as a whole in an exit form. Among the questions in the exit form, three of them were directly related to the subclass user interface:

1. I was able to use the sector hierarchy for my queries
2. The sector hierarchy was easy to navigate
3. Other comments about the sector hierarchy

For question 1 and 2, the users were asked to give points on a Likert scale, as suggested in [Hea09, Ch. 2]. In this case, a 7-point Likert scale was used, ranging from “Strongly disagree” (1) to “Strongly agree” (7). The third question was a free-text one.

Results for questions 1 and 2 are plotted in Figure 9.1. From this, we see that the responses with regard to the usability of the subclass interface were largely positive.

Question 3, asking for other comments about the sector hierarchy, was an optional free-text question. Nine out of the fifteen participants submitted an answer to this question. Their answers are listed verbatim in Table 9.4.

Answers from participants P6, P19, P20, and P25 show that they had a positive experience using the subclass interface. The rest of the
Table 9.4: Free-text answers to the question “Other comments about the sector hierarchy”

<table>
<thead>
<tr>
<th>ID</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>The key word suggestion was more useful. I haven’t tried with iskrem, i wonder now if it is possible but if not that would be a good thing. Otherwise the user needs to know the exact name of the sector.</td>
</tr>
<tr>
<td>P2</td>
<td>I discovered the sectors too late, this have made my results poorer then they could have been if I was able to find it earlier.</td>
</tr>
<tr>
<td>P4</td>
<td>The sector would be even better if you integrated synonyms, so you for example could search ”melk(eprodukter)” and still get the dairy sector.</td>
</tr>
<tr>
<td>P6</td>
<td>lovely!</td>
</tr>
<tr>
<td>P12</td>
<td>It would be really helpful to be able to ”backtrace” a sector. You can see which sectors are inside sector x, but not which sector x is inside.</td>
</tr>
<tr>
<td>P18</td>
<td>I wasn’t too bothered by it because the questions asked in the experiment made it obvious- I just typed in text to find them without using the hierarchies at all. However, if I was on my own and had my own queries, I might have a different impression of it.</td>
</tr>
<tr>
<td>P19</td>
<td>It felt good.</td>
</tr>
<tr>
<td>P20</td>
<td>It was great!</td>
</tr>
<tr>
<td>P25</td>
<td>NICE EFFORT AND GOOD WORK !!!!</td>
</tr>
</tbody>
</table>

answers deserve a more in-depth discussion.

Participants P1 and P18 seemed to prefer the free-text sector search over the hierarchical subclass menu. We realize that the free-text search might be simpler to use in many cases, as it frees the user from having to think about which classes to expand when looking for a particular subclass.

Participant P2 expected that his search performance would increase after discovering the sector hierarchy. This can be explained by the lack of preliminary training; the participants had to discover the subclass part of the interface themselves. It seems that participant P2 did discover the subclass hierarchy – albeit a bit late – and that the discovery increased his confidence with his search results.

Participant P4 suggested that aliases should be added for certain classes, as to make them easier to discover through the free-text search. This suggestion seems valuable, and may be implemented in future versions of the prototypes.
Participant P12 suggested that it would be useful to see which sectors a subsector belongs too. This is indeed possible when using the subclass hierarchy, by tracing the classes up through the foldable menu. The suggestion is valid if the class was found through a keyword search; in this case, it is currently not possible to see neither the sub- or superclasses of that class. Being able to do so would be nice, and we may choose to implement this feature in future versions of the prototypes.

### 9.4 Effectiveness of Subclass Interface

In addition to the users’ responses to each task, the SPARQL queries constructed by the tools during the user study were logged as well. By analyzing the raw queries, we were able to tell in which cases the subclass interface was actually used during query construction. This section gives an analysis of how the usage of the subclass interface affected the retrieval performance.

Note that in this analysis, we considered only those questions where we expected subclass reasoning to be relevant. Those were questions 1, 4, and 6, about the wireless telecommunication companies, accommodation establishments, and dairy companies respectively. Indeed, by looking at the constructed queries, we confirmed that the subclass functionality was never used for tasks other than these three.

We measured the retrieval performance of a user for one task as the F-measure of the submitted answers. Since the solution to each task in the survey was a list of companies, we calculated the precision and recall for each answer. In this case, the precision of an answer is the ratio of correct companies submitted to the total number of submitted companies. Recall is the ratio of correct companies submitted to the actual number of correct companies. F-measure is defined as the harmonic mean of precision and recall [MRS08, Ch. 8.3].

Now we would like to investigate whether use of the subclass interface affected the performance in the tasks where subclass hierarchies were relevant. We acknowledge that a number of factors might affect the performance of a particular task. To account for this, we performed an analysis of variance (or ANOVA for short), presented in detail in the following paragraphs.

ANOVA is a method for calculating how a number of factors affect the mean of a set of observations (see [Jai91, Ch. 15.1.1] for details). In this case, the set of observations is the performance (F-measure) of each user for each task. As mentioned in the introduction of this chapter, using the subclass interface was relevant in three of the tasks, and we only consider these observations in our model. As potentially relevant factors, we consider the version of the tool used, the particular task at
hand, and whether the subclass interface was used to solve the task or not.

The purpose of applying ANOVA here is to estimate how much of the variation in the performance is due to variation *within* each factor, versus the variation *between* the factors. We pose the following three null hypotheses, in hope that we may later reject them:

1. $H_{0A}$: the tool used does not affect the performance mean.
2. $H_{0B}$: the performance mean is the same across the different tasks.
3. $H_{0C}$: using the subclass interface does not affect the performance mean.

Applying the ANOVA procedure to our observation data, we may be able to reject any of the posed hypotheses. We used the ANOVA procedure `anova` in MATLAB\(^1\) to perform the actual calculations. Table 9.5 sums up the results.

We used here a linear ANOVA model. Other models can be applied, which take interactions between the factors into consideration. But every other model either gave a worse fit to our data, or could not satisfy the ANOVA assumptions [Jai91, Ch. 20.6]. Our current model passed the Lilliefors test [Lil67], and visual diagnostic tests.

Table 9.5: ANOVA table for performance on questions where subclass reasoning was relevant

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>Degrees of Freedom</th>
<th>Mean Square</th>
<th>F</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>0.2389</td>
<td>1</td>
<td>0.2389</td>
<td>3.3998</td>
<td>0.0742</td>
</tr>
<tr>
<td>Task</td>
<td>0.5865</td>
<td>2</td>
<td>0.2933</td>
<td>4.1728</td>
<td>0.0242</td>
</tr>
<tr>
<td>Used SI</td>
<td>0.4122</td>
<td>1</td>
<td>0.4122</td>
<td>5.8647</td>
<td>0.0211</td>
</tr>
<tr>
<td>Error</td>
<td>2.3192</td>
<td>33</td>
<td>0.0703</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>3.3857</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both the task factor and the subclass interface factors have $p$-values less than 0.05. The $p$-value is the probability of getting results as least as extreme as the observed ones, given that the null hypothesis is true. For instance, this means that assuming $H_{0C}$ to be true, the probability of getting the observations we did is 0.0211, or just above 2%.

If we decide to reject null hypotheses that have a probability of less than 5% to be true, we say that we assume a *significance level* of 0.05. That is, we deem $p$-values of less than 0.05 as significant results, and reject the corresponding null hypothesis.

\(^1\)http://www.mathworks.se/products/matlab/
The population marginal means of groups Used SI=0 and Used SI=1 are significantly different.

(a) Comparison of the performance means when using the subclass interface, versus not using it.

(b) Comparison of the performance means when using the subclass interface, versus not using it, combined with the task performance.

Figure 9.2: ANOVA result plots
Assuming a significance level of 0.05, we see that $H_{OB}$ and $H_{OC}$ can be rejected because both their $p$-values are below 0.05 (they are 0.0242 and 0.0211 respectively). That is, the performance differed significantly across the different tasks, and was significantly better when the subclass interface was being used.

Figure 9.2 plots the results of the analysis of variation. The plots were created using the `multcompare` procedure in MATLAB. Figure 9.2a compares the performance means when using the subclass interface, versus not using it. Figure 9.2b compares the performance means when using the subclass interface, versus not using it, combined with the task performance. Used SI=0 denotes that the subclass interface was not used, while Used SI=1 denotes that it was. Circles represent the group performance mean along the $x$-axis, while horizontal bars represent the confidence intervals. That is, we can assume that the difference in group means is significant when the confidence intervals don't overlap.

By comparing the performance means in Figure 9.2b task by task, we see that the performance increases when the subclass interface was used, but not significantly so when looking at single tasks at a time. From Figure 9.2a it is however clear that the difference in performance becomes significant when looking at the total mean across all the tasks.

With that, we conclude that using the subclass interface (when relevant) to construct queries increases the retrieval performance of the users.

## 9.5 Summary

We wanted to assess the usability of our approaches to visual query formulation for Linked Data, and particularly our implementation of subclass hierarchies in the user interface. This was achieved through a user study, where participants used our prototype tools to search the Semicolon dataset for information about Norwegian companies.

Their subjective opinions regarding the tools and their usefulness were recorded through online forms. The objective effectiveness of exploiting subclass reasoning in the user queries was collected by analyzing the constructed queries.

Even if some of the users remained unaware of the subclass hierarchies and how to use them, we perceived the overall feedback as mainly positive. The quantitative analysis showed that employing the subclass reasoning mechanisms significantly increased the performance of the users.

All in all, the survey has given us confidence that our current approach to the subclass interface is an acceptable one.
Chapter 10

Conclusions

Let us come back to the introductory example of the ambitious journalist who wanted to answer the following question:

Which Norwegian fishing companies have ever had a yearly revenue of more than 100,000,000 (one hundred million) Norwegian kroner, but no more than 10 employees?

Recall that to answer this question, the only real option the journalist had was to have a computer expert query the Norwegian Entity Registry by means of SPARQL, a formal query language.

With PepeSearch, the situation has changed. By using the graphical user interface of PepeSearch to formulate his query, the journalist would be able to answer the question on his own. A video depicting the whole query formulation process in PepeSearch is available at http://folk.uio.no/simenheg/pepesearch.webm.

Figure 10.1: Tiny, but successful, fishing businesses in Norway found by PepeSearch

As seen in the video, after selecting the “Organization” class, the user selects the “Fishing” sector from the hierarchical class menu, puts restrictions on the facets for income- and number of employees, and
retrieves the correct results almost instantaneously. The results of the query are reproduced in Figure 10.1.

Note that all the three companies in the result list belong to the sector “Marine fishing” (Norwegian: “Hav- og kystfiske”), even though the user searched for companies in the more general “Fishing” sector. Queries like this are made possible due to subclass reasoning.

In the remainder of this chapter, we will discuss the concrete contributions of this thesis and articulate our results, before discussing ideas for future improvements.

10.1 Contributions of This Work

Coming back to the goals for this thesis, we will relate them to the contributions made during the course of the work for this thesis.

In Chapter 5, we analyzed five different alternatives for supporting subclass reasoning in query formulation. As a case study, we implemented them for subclass reasoning over the sector hierarchy from the Norwegian Entity Registry in Chapter 6.

The different subclass reasoning alternatives were compared, and evaluated by their quantitative performance and qualitative characteristics in Chapter 7.

We contributed to the development of two search tool prototypes for Linked Open Data, and integrated mechanisms for supporting class hierarchies in these tools, both in the user interface, and in the query formulation process. We also contributed to a paper on PepeSearch, which is submitted for publication [Veg+14]. The SPARQL endpoint analyzing component constitutes our largest standalone contribution to the tools, and its source code is released on GitHub[^1]. The tools, and our contributions to them, were presented in Chapter 8.

We applied the search tool prototypes to several Linked Open Data sources, including the Linked Open Data version of the Norwegian Entity Registry. In Chapter 9, we evaluated how well our subclass interface performed by conducting a user experiment.

10.2 Results

Through the work for this thesis, we gained a number of insights which we will summarize in this section.

RDFS provided an excellent framework for subclass reasoning support, especially with a backward reasoning strategy. Even if

[^1]: [https://github.com/simenheg/sparql-endpoint-analyzer](https://github.com/simenheg/sparql-endpoint-analyzer)
forward reasoning can lead to maintenance problems when dealing with often-changing datasets, it provided good performance with respect to both absolute response time, and response time stability during our experiments in Section 7.1.

**Query rewriting** also provided comparatively good performance, while allowing us to perform subclass reasoning independently of the main data store. This is our recommended approach for supporting subclass reasoning when the main data store – for some reason – cannot be modified.

**Property paths** and **query federation** turned out to be unfeasible solutions to support subclass reasoning in our case for the Norwegian Entity Registry dataset. They provided comparatively bad, and highly variable response times in our performance analysis in Section 7.1. The query federation technology felt too immature to meet our demands for reliability, as elaborated in Section 7.2.

From the implementation and user study of our search tool prototypes, we learned that visual query formulation can indeed serve as a useful mechanism for searching Linked Open Data sets, even for people with no prior knowledge of Semantic Web technologies or formal query languages. Furthermore, we learned that users were able to employ a graphical class hierarchy interface in order to pose queries involving subclass reasoning. We saw in Chapter 9 that using the subclass interface significantly improved the users’ retrieval performance.

### 10.3 Improvements and Future Work

With respect to our study of approaches for supporting subclass reasoning, we don’t regard our list of investigated approaches as exhaustive, but we believe that we have covered some important, commonly occurring cases. Exploring new approaches – or testing other implementations of our existing ones – could be a fruitful exercise.

From the search tool prototype user study presented in Chapter 9, we received several useful suggestions with respect to improving the graphical subclass interface. Specific suggestions include making aliases for class names, making them easier to discover through free-text searches, and making it possible to see the hierarchical position of a class after selecting it through a free-text search.

Fifteen people participated in our user study, all of which were associated with the fields of either informatics or library science. Our tools turned out to perform well with this particular group of people, but even if this pilot experiment gave indications that our tools work well for querying Linked Open Data, more comprehensive user studies are needed to investigate how well our tools really work for the “man on the street”.

73
At the time of writing, PepeSearch has just been released to the public\(^2\), making it possible for anyone to adapt the tool to new datasets. We are excited to see how PepeSearch is going to work “out in the wild”, and whether it will inspire other people to adapt it to new domains, unpredicted by us.

Development of the SPARQL endpoint analyzer component continues at GitHub\(^3\). Future work for the component involves migrating to the mentioned JSON-LD\(^4\) output format, and continuing to test it against an even larger set of SPARQL endpoints to increase the robustness of the tool.

\(^2\)https://github.com/guiveg/pepesearch
\(^3\)https://github.com/simenheg/sparql-endpoint-analyzer
\(^4\)http://www.w3.org/TR/2014/REC-json-ld-20140116/
Appendix A

Prefix Table

The following table summarizes prefixes used in this document (as explained in Section 2.1.1).

Table A.1: Table of common RDF prefixes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inace</td>
<td><a href="http://data.computas.com/enhetsregisteret/nace/">http://data.computas.com/enhetsregisteret/nace/</a></td>
<td>Semicolon NACE instances</td>
</tr>
<tr>
<td>lok</td>
<td><a href="http://data.computas.com/informasjonsmodell/lokasjon/">http://data.computas.com/informasjonsmodell/lokasjon/</a></td>
<td>Semicolon location namespace</td>
</tr>
<tr>
<td>nace</td>
<td><a href="http://data.computas.com/informasjonsmodell/nace/">http://data.computas.com/informasjonsmodell/nace/</a></td>
<td>Semicolon NACE classes</td>
</tr>
<tr>
<td>org</td>
<td><a href="http://data.computas.com/informasjonsmodell/organisasjon/">http://data.computas.com/informasjonsmodell/organisasjon/</a></td>
<td>Semicolon organization namespace</td>
</tr>
<tr>
<td>rdfs</td>
<td><a href="http://www.w3.org/2000/01/rdf-schema#">http://www.w3.org/2000/01/rdf-schema#</a></td>
<td>Resource Description Framework Schema</td>
</tr>
<tr>
<td>unit</td>
<td><a href="http://data.computas.com/enhetsregisteret/enhet/">http://data.computas.com/enhetsregisteret/enhet/</a></td>
<td>Semicolon organization instances</td>
</tr>
</tbody>
</table>
Appendix B

SPARQL Endpoint Analyzer Manual

This is the manual for the SPARQL endpoint analyzer component of PepeSearch and friends: tools developed at the University of Oslo for simple exploration and querying of Linked Open Data sets.

The user interface of PepeSearch employs a configuration file in order to present the user with available concepts, links and properties. This program automates the task of creating that configuration file.

Full source code and binaries are available at https://github.com/simenheg/sparql-endpoint-analyzer.

B.1 Usage

Source- and binary versions are run in the same way:

```
./sparql-endpoint-analyzer.lisp settings.conf 1>output.js
```

This command reads the configuration file settings.conf, and puts the result into the file output.js.

B.2 Configuration

A configuration file for the SPARQL endpoint analyzer consists of 8 sections, described below. For a sample configuration file, see the next section.

- endpoint: URL of the SPARQL endpoint of interest.
• **hard-limit:** hard limit for each query sent, **before** filtering is done. This affects the number of retrieved concepts, literals, links and subclass relations.

• **page-limit:** maximum number pages to retrieve after getting a timeout, and attempting a paged retrieval.

• **results-per-page-limit:** maximum number of results per page during a paged retrieval.

• **prefixes:** list of predefined prefixes for convenient naming of JavaScript variables.

• **exclusive-whitelist:** list of URI prefixes. Results will **only** include URIs prefixed by a string in this list. An empty list disables this feature. When enabled, this feature overrides the normal black- and whitelist.

• **blacklist:** list of URI prefixes. Results will **not** include URIs prefixed by a string in this list, unless listed in whitelist. An empty list disables this feature.

• **whitelist:** list of URI prefixes. Results will **not** include URIs prefixed by strings listed in blacklist, unless they’re prefixed by a string in this list. Useful when one wants to whitelist particular URIs in a blacklisted domain.

### B.3 Sample Configuration File

The following configuration file can be used to slurp the Norwegian Entity Registry endpoint:

```plaintext
# Sample configuration file to retrieve information from the
# Norwegian Entity Registry (http://data.computas.com/)

[endpoint]
http://data.computas.com:3030/sparql

[hard-limit]
10000

[page-limit]
5

[results-per-page-limit]
10000

[prefixes]
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```
B.4 What Is Collected

The PepeSearch configuration file is in the JSON (JavaScript Object Notation) data format, which integrates naturally with the existing JavaScript code base. Four categories of data are collected in this file:

- Types
- Object properties
- Datatype properties
- Subclass relations

For each of these categories, further details are elaborated on in the sections that follow.

B.4.1 Types

Every type found in the dataset is recorded. That is, every `?type` matched by the following RDF triple:

```
?concept a ?type .
```
Types are mapped to concepts in the user interface. Together with its URI, each type entry also contains a short ID for convenience, a human-readable label with possible translations, the ID of a human-readable datatype property for use in the interface, and whether or not the type has any subtypes.

Example entry:

```json
{
    "id": "foaf_Person",
    "uri": "http://xmlns.com/foaf/0.1/Person",
    "label": {
        "en": "Person"
    },
    "display": "foaf_name",
    "primary": true
}
```

B.4.2 Object Properties

We define an object property as any RDF property linking two resources that have an RDF type. That is, every ?object_property matched by the following RDF graph:

```
?subject a ?subject_type .
?object a ?object_type .
```

Object properties are mapped to incoming- and outgoing links in the user interface. Objects become targets of the subjects’ outgoing links, while the subjects become target of the objects’ incoming links.

B.4.3 Datatype Properties

We define datatype properties as literals linked to by concepts via any property. That is, every ?literal matched by the following RDF graph, filtered by the `isLiteral` SPARQL predicate:

```
?concept a ?type .
```

B.4.4 Subclass Relations

Subclasses are defined by the `rdfs:subClassOf` property. That is, every ?subclass matched by the following RDF graph, where ?subclass ≠ ?class:
?subclass rdfs:subClassOf ?class .
References


