FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2017 Open / ~~Confidential~~
Author: Øyvind Blaauw	 (signature author)
Instructor: Prof. Krisztian Balog Supervisor(s):	
Title of Master's Thesis: Answering Engine for Sport Statistics: Question Processing	
ECTS: 30	
Subject headings: Semantic Web Linked Data RDF Knowledge base Question Answering Systems	Pages: 86 + attachments/other: Appendix (4 pages) + attached compressed file containing source code (1 zipped file) Stavanger, 15.06.2017 Date/year

Answering Engine for Sport Statistics: Question Processing

Øyvind Blaauw

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

June 2017

Acknowledgements

I would first like to thank my supervisor, Professor Krisztian Balog at the University of Stavanger. Professor Balog has shown both a great interest and knowledge of the field, and has provided me with interesting literature and feedback. Thank you for your guidance during the time of writing this thesis.

I would also like to thank my fellow student and thesis-partner, Aida Mehdipour Pirbazari. She has challenged me to think outside the box and given me useful feedback throughout the project. Thank you for your cooperation and your solid work with this thesis.

Finally, I want to express my deepest gratitude to my family, my wife Stephanie and my son Aron. You have motivated me with kind and supportive words, laughter, encouragements and handled other tasks so that I could work with the thesis. Without your unfailing support, this accomplishment would not have been possible.

Thank you.

Abstract

In recent years, there has been an increasing growth of interest among computer scientists for the topic of Linked Data and the Semantic Web. By connecting and publishing structured data from multiple sources, the Web enables us to retrieve specific information without needing to go through documents of unstructured text. Question answering systems can utilise the benefit of Linked Data, and enable users to ask question in a natural language in order to provide direct answers. In this thesis we implement a system that can answer natural language questions related to the field of Formula 1 statistics. We show how data is collected and connected based on a conceptual model, and go through the necessary steps for converting a question into a machine-readable query. We perform an evaluation of the system, both on component level and on the system as a whole. We analyse and discuss challenges and topics for improvements, before we conclude our work and summarise the most important steps to consider for future work.

Contents

1 Introduction **8**
1.1 Motivation . . . 8
1.2 Project objectives . . . 9
1.3 Outline . . . 11

2 Background **12**
2.1 Formulating and Answering Information Needs . . . 12
2.1.1 The Semantic Web . . . 14
2.1.2 Linked Data . . . 15
2.1.3 RDF . . . 17
2.1.4 Ontology . . . 21
2.1.5 Knowledge Base . . . 24
2.1.6 DBPedia . . . 25
2.2 Question Answering Systems . . . 25
2.2.1 History . . . 25
2.2.2 Related work . . . 27
2.2.3 Question Classes . . . 28
2.3 Overview of QA terminology . . . 30
2.3.1 Question Types . . . 31
2.3.2 Question Phrases . . . 31
2.3.3 Answer Types . . . 31
2.3.4 Question Focus and Topic . . . 32
2.3.5 Authority Lists . . . 32
2.4 Natural Language Processing . . . 33

3 Implementing a QA System **36**
3.1 Ontology . . . 36
3.2 Knowledge Base & Lexicons . . . 40
3.2.1 Data collection . . . 40
3.2.2 Tools . . . 40
3.2.3 Lexicons . . . 40
3.3 Approach . . . 42
3.3.1 Phrasing . . . 43
3.3.2 Determining answer type . . . 45

3.3.3 Semantic Phrase Detection and Mapping 46
3.3.4 Named Entities 46
3.3.5 Query generation 48
3.4 Full example 51
3.4.1 Phrasing 52
3.4.2 Answer Type Detection 53
3.4.3 Semantic Phrase Detection 53
3.4.4 Semantic Phrase Mapping 53
3.4.5 Named Entity Recognition 53
3.4.6 Named Entity Disambiguation 54
3.4.7 Query Generation 54
3.5 Prototype of Web Interface 55

4 Evaluation **57**
4.1 Methodology 57
4.1.1 Baseline questions 57
4.1.2 Ontology and KB Evaluation 59
4.1.3 End-to-end Performance 60
4.1.4 Analysing the components 61
4.2 Results 63
4.2.1 Knowledge base statistics 63
4.2.2 Ontology 65
4.2.3 End-to-end Evaluation 66
4.2.4 QA Components Evaluation 68
4.2.5 Error Analysis 70

5 Analysis and Discussion **72**
5.1 Precautions 72
5.2 Baseline Questions 73
5.3 Lexicons 73
5.4 Components 74
5.4.1 Phrasing 74
5.4.2 Answer Type Detection (ATD) 74
5.4.3 Semantic Phrase Detection (SPD) 75
5.4.4 Semantic Phrase Mapping (SPM) 76
5.4.5 Named Entity Recognition (NER) 76
5.4.6 Named Entity Disambiguation (NED) 76
5.4.7 Query Generation (QG) 77
5.5 General 78
5.5.1 Approach 78
5.5.2 Prototype 78
5.5.3 Evaluation 79
5.5.4 Overall goals 79

6 Conclusion **81**
6.1 Summary . . . 81
6.2 Future work . . . 82

Bibliography **83**

A Baseline Questions **87**

B Overview of Attachments **90**

List of Tables

2.1 SPARQL example results, displaying two variables 20
2.2 SPARQL example results, displaying one variable 20

3.1 Table of namespaces and prefixes used in the ontology 37
3.2 Look-up table of questions phrases, mapped to internal answer types 45

4.1 Categories and types of baseline questions 58
4.2 Table of Question Evaluation Categories (QEC) 60
4.3 Results table of end-to-end evaluation 66
4.4 Overall Accuracy 68
4.5 Precision, recall and F1-score of NER and NED 69
4.6 Types of errors and the related frequency of questions 71

List of Figures

1.1 System model 11

2.1 Snapshot of a Google-search for New York Weather 13
2.2 Linking Open Data cloud diagram 2017 [1] 15
2.3 Example of a RDF graph 18
2.4 Example of a simple ontology 22

3.1 Formula 1 ontology used for the QA system 37
3.2 Excerpt of resource-lexicon 41
3.3 Excerpt of phrase-lexicon 42
3.4 Pipeline of Question Processing Module 43
3.5 Result of words chunked to phrases 53
3.6 Web interface with a sample question 55
3.7 Presenting the results 56

4.1 Extraction of baseline questions 59
4.2 Distribution of class instances 64
4.3 Drivers and races per season 65
4.4 End-to-end evaluation for question types, distributed on QEC . . 67
4.5 End-to-end evaluation for factoid and list questions, distributed on QEC 67
4.6 Accuracy per Question Type 68
4.7 Overall Performance of QA components 69

Chapter 1

Introduction

1.1 Motivation

The amount of information and data being published on the Internet increases every day, including articles, news, multimedia, tweets, blogs, encyclopedia pages and more. Individually, every web page is considered as a document on the Web. Today, it's common for web pages to link to other sites through hyperlinks, and the entire collection of documents are referred to as *the Web of Documents*. In 1999, the World Wide Web Consortium (W3C) introduced the RDF (Resource Description Framework) framework which made it easier to store data in a more structured manner [25]. With RDF it was possible to link related datasets together, even though the underlying data models were in different formats. The Web of Documents slowly emerged into *the Web of Data*. In 2006, Tim Berners-Lee [36] introduced a set of guidelines to use for linking data using the RDF framework. This marked the beginning of Linked Data. The overall goal with Linked Data was to publish structured data that could be useful for *semantic queries*, meaning that data can be retrieved as useful information in stead of as blocks of "meaningless" text. Because datasets consisting of Linked Data hold meaningful and informative information, these databases are also called knowledge bases (KB).

The Linking Open Data (LOD) community project (also called Linked Open Data Cloud) was introduced shortly after the beginning of Linked Data. The goal was to encourage other computer scientists to increase the amount of Linked Data and interconnect knowledge bases, in order to create one big cloud of semantic information. According to the current LOD statistics[1], the cloud consists of nearly 150 billion RDF triples from almost 3000 datasets.

As more Linked Data is made available, the demand of acquiring and locating this information increases as well. However, one of the main challenges with Linked Data is to retrieve the information for the common web user. First of all, there are so many KBs available, and it's not trivial to figure out what KB

[1] `http://stats.lod2.eu/`

that holds the information the user is looking for. Second, the user would have to learn a query language to retrieve RDF data (SPARQL), in addition to all the domain-dependent vocabulary used in the data model. Through history people has gone from using hyperlinks in documents to web search engines, which made it easier to find information through keyword queries and text search. Search engines are within the concept of Information Retrieval (IR) systems, which retrieves and rank relevant documents from a large corpus. Today there exists many types of search engines designed for different purposes. However, even though some search engines like Google have started to include Linked Data in search results, most of them don't consider data from the LOD cloud. In most cases, this means that people are left with the task of finding the needed information from the returned documents.

One solution to this problem is the concept of Question Answering (QA) systems. In general, QA systems allows people to get access to data through natural language, which means that there is no need for a technical background or an insight to the underlying data model. They differ from common keyword-based search engines by enabling more complex, fully written questions. In addition they can interpret the request up against the available data and return a precise answer. Historically speaking, QA systems have been around for a long time and originally used text as data sources. Back in the early 1970's, QA systems also interacted with structured databases through natural language interfaces [31]. The performance and accuracy of QA systems have though increased since then. WolframAlpha[2] is one of the modern times' popular and powerful *knowledge engines*, and can interpret queries related to many different fields. In recent years, Question Answering over Linked Data has received more attention because of the great expansion of Linked Data, and is considered as an interesting and growing field of information research.

In this thesis we present a first-version implementation of a QA system over Linked Data developed from scratch. The purpose of this research is first of all to learn and further provide an insight to the exciting field of both Linked Data and QA. Second, we review some of the methods already used in QA systems today and use some of the more common procedures for our own implementation. The goal of this research is not necessarily meant to be an introduction to an improved and revolutionary approach, but it would be considered as a bonus if readers are encouraged to implement their own systems and contribute to the cloud of Linked Data. However, in order to be innovative and come up with new solutions, it's important to cover the basics and start from the ground, especially in a field that can be complex to understand in the beginning.

1.2 Project objectives

The project was presented by the University of Stavanger as a candidate assignment for a master thesis. The objective of the project was to build an answering

[2] `https://www.wolframalpha.com/`

engine that could interpret natural language questions related to sport statistics. The specific tasks included

- Collecting data or connecting to existing data services
- Building a conceptual model (ontology) of the chosen domain of sports
- Developing a question interpreter that can understand concepts related to the selected field of sports (names of players, teams, leagues, years etc.) and related to statistics (e.g. "best", "most", "highest", "on average" etc.)
- Developing an answering module that can process the interpreted question against a knowledge base of facts, and in addition to showing the answer, show the underlying data that was used for the calculations.
- Implementing a working prototype
- Performing an evaluation of the system

The project has been a cooperation between two students, myself and Aida Mehdipour Pirbazari, and the sport Formula 1 was selected as the domain. The thesis by Pirbazari will focus on collecting data and building the conceptual model. The paper presented here focuses on interpreting NL questions and converting them into queries that can be executed on the KB. Building the prototype and performing the evaluation have been a shared responsibility. However, the evaluation presented in this paper focuses on the performance of the question processing module, whereas the evaluation presented by Pirbazari focuses on the ontology design and the knowledge base. Figure 1.1 shows a complete overview of the system. The modules *Question Processing (QP)* and *Lexicons* are contributions of this thesis. *Answer Extraction* and *Ontology Construction* are covered by Pirbazari. Development of a GUI (Graphical User Interface) has been a shared task.

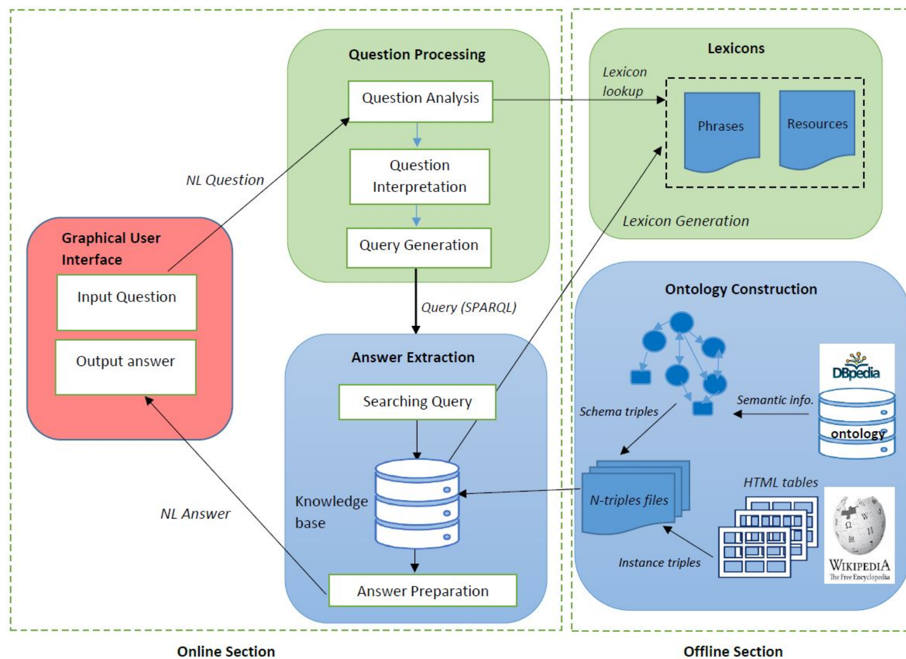

Figure 1.1: System model

1.3 Outline

The rest of the paper is structured as followed: Chapter 2 provides a detailed background of many of the topics already introduced, like Linked Data, RDF, knowledge bases, historical QA systems, techniques related to Natural Language Processing (NLP) and common terminology used in QA systems. Chapter 3 describes the implementation process of a QA system over Linked Data for the domain of Formula 1 statistics. The overall goal with this implementation is to develop a QA system that can answer NL questions, based on a KB that has been developed from scratch. In Chapter 4 we evaluate the system and present the results. In Chapter 5 we discuss some of the challenges we met, in addition to the strengths and weaknesses of the system. We conclude our work in Chapter 6.

Chapter 2

Background

2.1 Formulating and Answering Information Needs

Search engines have for a long time operated around user queries consisting of keywords, like "university Stavanger". This method of information retrieval is convenient when the user knows what to look for and which keywords to include because it's fast and intuitive. In addition, this serves as a good method when searching for various information about some topic, where several sources may be needed. This kind of searching are sometimes referred to as a lexical approach, where the keywords typed by the user, or some variant or synonyms of them, are literally matched up against a large corpus [2]. There is no further understanding of the query other than literal matching. Once the search engine returns a list of relevant pages or documents, the user must decide which sites are relevant for the information he or she is looking for. However, not knowing how to formulate a question, what to look for or which websites to visit makes the information retrieval process much more time consuming and impractical. Including either too few or too many keywords will affect the quality and quantity of the search results.

Considering the example above, 'University Stavanger', the search engine will match the keywords to relevant documents or sites by looking at text passages, URLs, titles and other texts in the documents, before returning a ranked list - usually in terms of hyperlinks to the hosting websites. Getting to the homepage of the University of Stavanger (UiS), the Facebook page of UiS or even the Wikipedia page of UiS are all good matches for general-purpose information, but the search engine does not have any comprehension of the meaning, or the semantics, of the terms 'university' and 'stavanger', or what they refer to together. Nor does it have to. It's simply about matching some keywords to a large corpus of text, and letting the user decide the meaning of it.

Modern search engines have been and are still expanding their capabilities when it comes to understanding search terms. When using Google[1] to search for

[1] www.google.com

"New York weather", though still returning a list of relevant sites for weather forecasts, Google interprets or "understands" this query as "How is the weather in New York City, USA today, and what does the forecast look like the next seven days?", including cloudiness, temperature, rainfall, humidity and more. The results are presented in a nicely formatted table at the top of the page, as shown in Figure 2.1, making it user-friendly.

Figure 2.1: Snapshot of a Google-search for New York Weather

It's even possible to write more specific questions like "How is the weather in LA the next week?", Google will still be able to detect the meaning of the question in this example. However, even though there is a high probability that the interpretation above was the intended one, it's also possible that the user was merely interested in some historical or general information about New York weather, and not the current forecast. Strictly speaking, these kinds of queries should perhaps include the terms 'general' or 'history' in addition. But the ideal thing would be to tell the search engine the exact information to look for, and retrieve the appropriate answer - like in the weather example above.

This is however not always the case of search engines, where machines and logical software have the responsibility of detecting the meaning of natural language questions or queries. There are several reasons for this. First of all is disambiguation, or word-sense disambiguation, which is the process of identifying the meaning of a word used in some context [27]. This includes both understanding general language as well as detecting *entities*, i.e. an existential thing, subject or object like a specific person or a place. An example of everyday-language is the term 'play', which can be referred to as children play-

ing, a theatre production, playing for a team or acting in a movie. It could even refer to some specific movie called 'Play', as well as many other things. Determining the meaning of this word is dependent on the context it's used in, but even then it's still a challenge for computers to map it to some relation or resource.

A second challenge is that the information has to be available and preferably linked together. The World Wide Web contains countless documents and pages referring to different topics, distributed on numerous servers and databases across the globe. There is no standardisation of linking related information, which is usually done by the use of hyperlinks. This will be further discussed in Section 2.1.2. Retrieving information and answering questions about e.g. the Formula 1 racer Lewis Hamilton would be simple (or simpler) if all possible information about that driver was located in the same database. However, this is usually not the case even for such a relative small domain. It's definitely not the case when considering an open-domain system consisting of all the information available on the Web, or a large amount of different domains.

Researchers and scientists have for many years, especially since the introduction of TREC as described in Section 2.2.1, been working on linking data together in order to build *The Semantic Web*[2].

2.1.1 The Semantic Web

The Semantic Web is often confused with the concept of Linked Data, and there are different opinions about this comparison today [29]. However, most researchers within this field considers the Semantic Web as a vision of a Web consisting of Linked Data (a Web of Data), meaning that it's made up of structured data gathered and linked from multiple sources. It's also compared to the Linking Open Data (LOD) community project, which is a specific project that concretises the vision. Supported by W3C, the primary goal of this project is to expand the Web of Documents with the Web of Data, representing data on a large scale in a format that is similar to such found in databases [35]. This will make it easier for computers to retrieve specific information that can be collected from structured data, which again is an important factor for enabling *semantic queries*. In this context, semantic queries refers to questions, keyword or text search with meaning - for example looking up a fact. To sum up, the purpose of the semantic web is to make computers understand the meaning of data, making it easier to exchange information with humans [33].

The process of building the Semantic Web is reliant on having as much data as possible available. More importantly, data has to be available in a structured and standard format, so it can be reachable and manageable by semantic web tools [33]. In addition, it must be possible to create a relation between different datasets, or link them together. These concepts are covered by Linked Data, described subsequently.

[2] `https://www.w3.org/standards/semanticweb/`

2.1.2 Linked Data

The Internet and the World Wide Web revolutionised data accessing, information retrieval and communication. Anyone can contribute by publishing data, documents, knowledge and personal opinions, making the Web one big source of global information. However, it's not necessarily easy to go through this data and locate useful information. One method is to use a search engine and rely on its indexing approach and similarity metrics in order to get the most appropriate and relevant data from web documents. Another way is to follow hyperlinks in articles, web pages and encyclopedias that, usually, have been manually inserted by someone in order to find other related data. Though worth mentioning, this technique of digging deeper is already implemented by many of the popular search engines today.

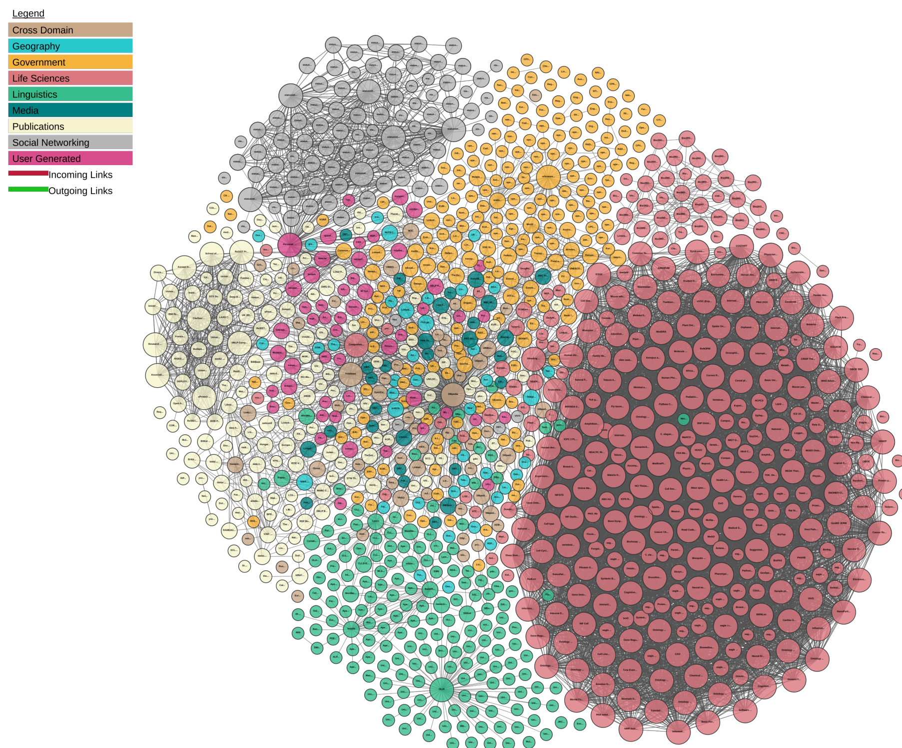

Figure 2.2: Linking Open Data cloud diagram 2017 [1]

It's also possible for people and organisations to give public access to databases that holds useful information. But this requires people to learn many different kinds of data models, in addition to a potential query language to retrieve the ac-

tual information. Even if there were user-friendly interfaces implemented, they would still face the same challenge: there are multiple sources of information in different formats.

Linked Data (LD), as the term implies, refers to data that are linked together. More precisely, it is a set of approaches and best-practices to connect structured data on the Web, creating a *Web* of Data as opposed to a *collection* of data [33]. There are mainly two important things that are necessary for creating LD. First of all, the data needs to be converted into a standard, common format so it can be reachable and manageable [33]. Second, it must be possible to link data from different sources so that relevant information can be gathered. The overall goal by implementing LD is first to make information machine-readable, meaning that computers can understand what the information represents, and second, enable semantic queries so that people can access and retrieve meaningful information without needing to go through multiple documents.

As discussed in the previous section, LD is a fundamental factor for realising the Semantic Web. Figure 2.2 gives an insight on how the LOD cloud looks like today. But LD is also useful outside this context. Companies might use LD to publish information about their products to their customers, enabling them to search for product specifications. The company can also link to the vendor's dataset for a specific brand in order to give more information about a product.

By using a standard format for all data, LD makes it possible to merge different data models that previously has been difficult to do on a technical level. Web pages primarily use the markup language HTML (HyperText Markup Language) to structure and format data. LD on the other hand, mostly relies on the RDF framework that was adopted as a recommendation by W3C in 1999 [25]. With RDF it's possible to create typed statements that can link anything in the world [4]. RDF is further described in the next section.

In 2006, Tim Berners-Lee, often recognised as the inventor and today the director of the World Wide Web, introduced a set of guidelines to use for linking data using the RDF framework. This set of rules is known as the "Linked Data principles". The purpose was to provide a guideline on the way of creating the Web of Data so that all contributions would follow a common standard and structure. The following principles are retrieved from [33, 4]

1. Use URIs (Unique Resource Identifier) 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 standards like RDF/SPARQL)

4. Include links to other URIs, so that they can discover more things

As discussed, the overall goal of connecting data in this way is to make relevant, published data machine-readable. This means that a machine should be able to look up, read and understand the information [4]. To do this, it must

also be possible to define the meaning of the data, e.g. what does it describe and which properties does it have. This is covered by RDF.

2.1.3 RDF

The Resource Description Framework (RDF) was originally a group of specifications from W3C, used as a model to represent data about data, or metadata. Today it's referred to as a generic, graph-based data model commonly used for describing resources in conceptual models, similar to the classical entity-relationship models [25]. More specifically, the RDF data model is used to describe web resources through statements, known as RDF triples. These triples are expressed in the format *Subject - Predicate - Object*, where the subject represents a resource, the object could be a resource or a literal value, and the predicate represents the link or relation between the subject and the object. For example, to express that Charles Dickens wrote the book Oliver Twist, the following RDF triple can be constructed:

Subject: Charles Dickens
Predicate: wrote book
Object: Oliver Twist

RDF makes it possible to exchange data on the Web, even though the underlying data models are different from each other. The framework uses URIs to uniquely identify resources as well as predicates. This allows both structured and semi-structured data to be mixed and shared across platforms, by merging the data into a heterogeneous representation [34]. The following triple demonstrates how the example above can be expressed using (HTTP) URIs:

Subject: `http://example.org/resource/Charles_Dickens`
Predicate: `http://example.org/ontology/wroteBook`
Object: `http://example.org/resource/Oliver_Twist`

As discussed in the previous section, the RDF model is used (amongst other) for creating Linked Data. The example above actually demonstrates all of the Linked Data principles by Lee; There are URIs used as names for things, which again are HTTP URIs so that people could look up those names. Third, the triple provides some "knowledge", or useful information about both the writer Charles Dickens as well as the novel Oliver Twist. When looking up the resource (URI) Charles Dickens, the triple above also includes a link to the URI of the novel Oliver Twist.

The example triple above could fit into a number of different knowledge domains, e.g. English literature, the life of Charles Dickens, information about books for a specific library and so on. However, as more and more data is added to a domain, it quickly becomes difficult to read RDF triples in order to get an overview of the domain. The benefit of using the RDF model is that the data can be viewed as a directed, labelled graph. A graph is visually a better

representation of data as opposed to a large amount of text. The graph edges represent the named links (predicates) between two resources, which in turn are represented by the graph nodes [34]. Figure 2.3 demonstrates an example of such a graph.

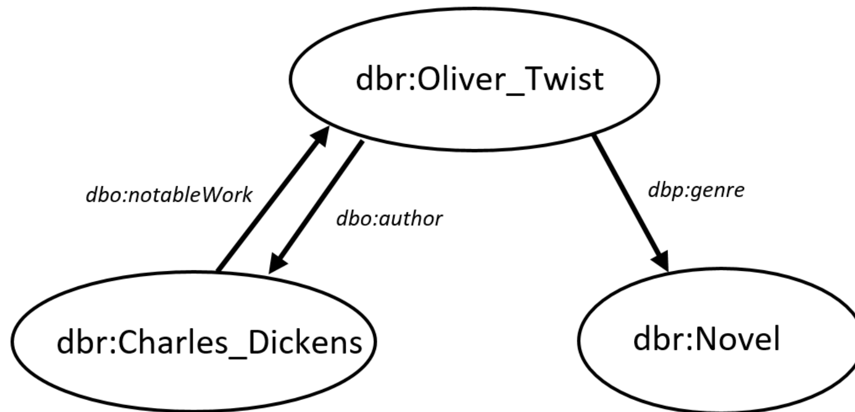

Figure 2.3: Example of a RDF graph

The vocabulary used in this figure, i.e. the names of the resources and predicates, are reused from the knowledge base DBPedia[3]. Note that there are prefixes used in stead of writing the full URI. For example, the prefix `dbo` refers to the DBPedia ontology URI, `http://dbpedia.org/ontology/`. It's often easier to read URIs with prefixes and especially the graphs are presented in a cleaner format without the full URIs.

Formats

RDF is an abstract model with several serialisation formats. Some of the most common ones are listed below, extracted from [25].

> **Turtle:** A compact, user-friendly format
>
> **N-triples:** A simple, line-based format that is not as compact as Turtle. Each line consists of a RDF triple, and the triples are separated by dots (.)
>
> **N3:** Similar to Turtle, but includes other features such as defining inference rules
>
> **RDF/XML:** The first standard format used for serialising RDF, based on the XML-syntax.

[3] `http://dbpedia.org/`

RDF triples are usually stored in databases specifically designed to handle RDF data, called *triplestores*. Related to the field of Linked Data and The Semantic Web, these databases are often referred to as knowledge bases, discussed further in Section 2.1.5.

SPARQL

Equally important as to having data available in a database or a triplestore, is to be able to retrieve the data. SPARQL is a query language used to retrieve and manipulate RDF data, and has a similar structure as SQL. Even though there exist other RDF query languages, SPARQL is perhaps the most popular and dominant query language used today [24].

A typical SPARQL query consists of one or more triple patterns. These patterns may include specific resources and predicates, but they can also include variables. Just like with SQL, the queries can include logical conjunctions (*and*), disjunctions (*or*), optional patterns, limitations and ordering. An example is demonstrated below.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT ?title
WHERE {
    ?book dbp:author dbr:Charles_Dickens .
    OPTIONAL {?book dbp:name ?title} .
}
ORDER BY ASC(?title) LIMIT 3
```

Listing 2.1: SPARQL Example

The example above demonstrates a simple SPARQL query that retrieves the titles of all the books written by the English author Charles Dickens, orders them in alphabetically order and returns the three first books. The information is collected from the DBPedia knowledge base. In order to avoid having to write the full URIs of resources and predicates, the query defines prefixes using the keyword `PREFIX` as shown in the two first lines of the query.

SPARQL variables are defined using the question mark followed by the variable name, such as `?title` and `?book`.

RDF triple patterns are listed inside the WHERE-clause, and act as the conditions of the query. Any triple from the knowledge base that matches *all* the conditions will be returned as a result. The SELECT-variables decides which part of the returned information will be displayed as the final result. This will become more clear when the results are presented. First, consider the two triples in the example.

The first pattern looks for any triples that has the predicate `dbp:author` and the object `dbr:Charles_Dickens`. The subject of this triple is a variable

?book	?title
`http://dbpedia.org/resource/A_Child's_History_of_England`	"A Child's History of England"@en
`http://dbpedia.org/resource/A_Christmas_Carol`	"A Christmas Carol"@en
`http://dbpedia.org/resource/A_Tale_of_Two_Cities`	"A Tale of Two Cities"@en

Table 2.1: SPARQL example results, displaying two variables

?title
"A Child's History of England"@en
"A Christmas Carol"@en
"A Tale of Two Cities"@en

Table 2.2: SPARQL example results, displaying one variable

called `?book`. This means that the query will collect all the triples satisfying the conditions of the predicate and the object, and collect all the subject resources in a table column under the variable `?book`. To sum up, this triple pattern collects all resources which has a property/predicate called author with the corresponding object value Charles Dickens.

The results of the first triple is a list of (resource) URIs. The second triple looks for literal titles of these resources, by looking for the object values of the predicate `dbp:name`. However, this is an optional triple as defined by the keyword `OPTIONAL`. This means that if it's possible to locate the title, then collect it under the `?title` variable. If not, then simply collect an "empty" slot in stead.

The final part of this query consists of ordering the results on titles in alphabetically order (ascending), and limiting the results to only show the three first rows (LIMIT 3).

Also note that even though there were two variables defined, only one of them was chosen as a SELECT-variable (`?title`). This means that data that were stored under the `?book` variable will not be shown in the final results.

Table 2.1 shows how the results would look like if the query included both `?book` and `?title` as SELECT-variables. Table 2.2 displays the results using only the `?title` variable.

2.1.4 Ontology

An ontology is perhaps one of the most important things to consider when it comes to organising linked data. It acts as a conceptual model that describes how data is structured and linked together. Today there exists numerous research papers on ontologies, how to design them and what rules to follow. This section presents the basic idea behind an ontology used for Linked Data and some of the fundamental principles behind it. This presentation will only scratch the surface of ontologies and will not go deeper into analysing ontologies, discussing the design or challenges of them.

What is an ontology?

In general, an ontology can be considered as a model on how data for a given domain is structured. It does not include actual data, but simply describe which kind of data that's available in the domain and how the data is connected in the big picture. It does this by using a hierarchy of classes and properties, which will be examined in more detail below. On the other hand, an ontology together with instances of data is called a *knowledge base*, and will be covered in the next section [22].

Another aspect of ontologies is that they describe a vocabulary. This is important when it comes to sharing knowledge with other researchers and scientists, and specifically for linking datasets. The reason is that an ontology, and also the corresponding knowledge base, uses a vocabulary that is mostly defined for that specific domain alone. The "vocabulary" in this context refers to the names used for different resources and predicates. For instance, the term "label" can in one ontology refer to the same concept as the term "name" in another ontology. Knowing that these terms represent the same property is important when it comes to connecting data from different ontologies - which again is the primary concept of The Semantic Web or the Linked Data Cloud.

An ontology is often best understood and reviewed when it is shown visually. For the remaining part of this section, the terminology and concepts of ontologies will be described using the ontology example in Figure 2.4 below.

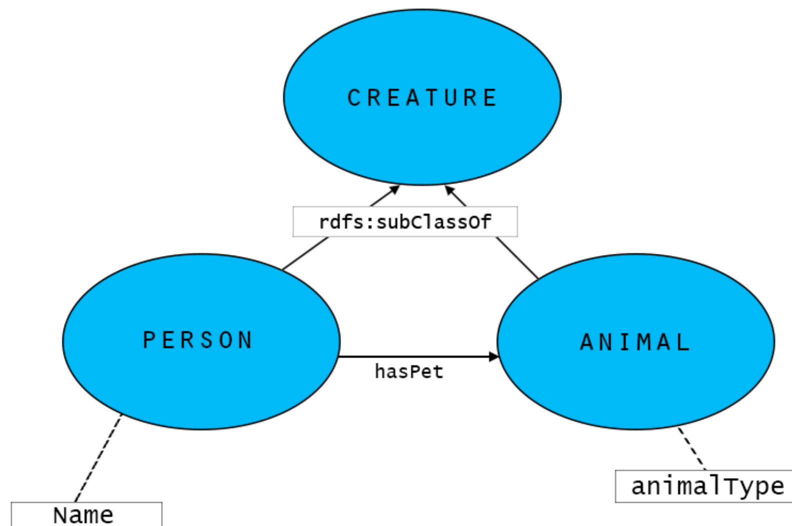

Figure 2.4: Example of a simple ontology

Classes

One of the most important things an ontology describe are *classes*. They describe which kind of data the domain covers on a higher level. In Figure 2.4 there are three classes presented; `Person`, `Animal` and `Creature` (names in `this font` refers to ontology vocabulary). `Person` describes the concept of people and `Animal` represents all animals.

Classes can also have subclasses, similar to the concept of inheritance in object-oriented programming. Generally, subclasses represents more specific concepts than the parent-class, or *superclass*. In this example, `Person` and `Animal` are subclasses of the superclass `Creature`. This is of course just one example on how to structure these classes, in the end it really depends on what relations the application or model needs to cover. Perhaps it is more necessary to introduce subclasses of animals, like dogs, cats, fish, birds, etc. Another taxonomic hierarchy is dividing the animals into mammals and non-mammals.

Properties

The ontology classes give a description of the concepts covered in the domain [22]. *Properties* on the other hand describe the classes. These properties can describe things about the class, but also the class' relation to other classes. Properties are also referred to as *slots*, and one principle often used when constructing an ontology is to include restrictions on these slots. This means that the property values must be of a certain type. For example, a property called *name* would typically have a value of type string. A birth date should require a type of date as the value, and so on.

As illustrated in Figure 2.4, properties are (often) illustrated with arrows in graphs. `Person` and `Animal` both have an arrow pointing towards `Creature`. This means that they both have a property, in this example called `subClassOf`, where the value is the class `Creature`. It's also possible to define properties in the opposite direction that describe the same relation. For example, `Creature` could have a property called "hasSubClass", with arrows pointing to both `Person` and `Animal`. Subclasses can have properties of their own if they are only of interest for that subclass. If the property is relevant for all the subclasses, then it can be defined at the superclass.

Another property defined in this ontology is `hasPet`, a property defined for `Person`. This shows that it's also possible to have properties that define relations with other classes, but not only in a hierarchical way (superclass-subclass). It's also possible to add multiple relations between two classes. For instance, a person can "have a pet" of type animal, a person can "eat" types of animals or a person can "hunt" different types of animals. Again, it depends on what the ontology is supposed to describe, the knowledge domain, that decides what properties to add.

Related to the RDF framework, properties are similar to predicates. A class can be both the subject and object of a triple. For instance, consider the two RDF triples below.

```
class:Person - rdfs:label - "Person"
Jane_Doe - rdf:type - class:Person
```

Intuitive enough, this means that classes can have predicates that describe the class, such as a label, hence being the subject of a triple. But, and quite often, the class can be the object of a triple. Instances in the knowledge base, such as the example of the person "Jane Doe", are defined with properties that describe which classes it belongs to. There are often more than one class an instance can relate to.

Web Ontology Language (OWL)

Classes can also have properties defining that they are a type of class, where a class is defined in the Web Ontology Language (OWL). Together with the RDF Vocabulary Definition Language (RDFS), they provide a basis for creating vocabularies [4]. Both OWL and RDFS are specifications from W3C [15], and W3C define OWL as "a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relation between things" [37].

Summary notes

It's important to create a solid foundation by putting a lot of effort and time into developing a good ontology. This is especially important if there is a plan of expanding the ontology. Having a poor design from the start will make it much more difficult to include other classes, subclasses and properties in the

future and in general expand the model. It's also necessary to think about what kind of knowledge that the ontology is supposed to provide. If constructing an answer or a fact based on the ontology is very difficult, it might point to a poor design.

Even though the ontology example presented here consists of only three classes and a few properties, it's easy to imagine all the properties that could have been added. Adding just a few more classes can expand the model even more quite rapidly.

As mentioned in the introduction of this section, there are many papers trying to explain ontologies, some are for experienced users and some are guides for researchers who are new to the field. One of these papers is *Ontology Development 101*, written in 2001 by Noy and McGuinness. They summarise some key steps of developing an ontology [22], listed below:

1. Define the classes in the ontology
2. Arrange the classes in a taxonomic hierarchy (superclass-subclass)
3. Define slots (properties) and describe allowed values for these slots
4. Fill in the slot values for instances

Once the ontology is ready, a knowledge base can be constructed by defining individual instances of the classes, and finally filling in the values for the properties.

2.1.5 Knowledge Base

In general, a knowledge base (KB) can be thought of as a repository of information and is quite comparable to a database. The purpose of a KB is to store information in an organised and structural way so that it becomes machine-readable. For Linked Data, the KB is often based on an ontology, consists of RDF data and enables retrieval of data through SPARQL queries. It holds concepts, classes and properties and is constructed according to the rules and requirements of the ontology. The difference between an ontology and a KB is that an ontology represents the conceptual model, whereas the KB is populated with instances according to this model. The term *knowledge* is often used instead of data (as in databases) because the goal is to gather and present actual *knowledge* about something, though there are often abstract text represented in a KB that strictly speaking may not be knowledge. A KB can combine data from different sources into a heterogeneous representation.

There are various purposes for using a knowledge base. A company might want to create a space that holds all necessary information about some product(s), so that their clients can look up information anytime with just a few clicks. Other organisations develops public knowledge bases, contributing to the Open Linked Data project (or the semantic web), such as DBpedia [9] and YAGO [42]. The DBPedia knowledge base consists of structured data extracted

from Wikipedia and can describe 4.58 million things [9]. YAGO extracts information from Wikpedia as well as other sources. DBPedia and YAGO are also linked together in order to integrate with the linked data cloud [42].

As described in Section 2.1.3, data can be retrieved from KBs through SPARQL queries (or other RDF query languages). The results are typically displayed as either graphs or as sets (tables).

2.1.6 DBPedia

DBpedia can also be considered as a large community project of building a Web of data. The project consists of extracting information from Wikipedia and making it available as structured data on the Web, i.e. the DBpedia knowledge base. DBpedia allows for sophisticated queries against Wikipedia content, and the English version can today describe up to 4.58 million things. Because DBpedia can describe so many things and has defined millions of URIs and concepts, it naturally becomes a central hub in the Web of Data to use for LD-based applications such as question answering systems [9].

2.2 Question Answering Systems

2.2.1 History

In 1992, NIST (National Institute of Standards and Technology) hosted the first Text Retrieval Conference (TREC) together with the U.S. Department of Defence. The purpose of TREC was to gather the community of Information Retrieval (IR) research, provide support, resources and the necessary infrastructure for evaluating different kinds of methods and implementations of IR systems. In 1999, TREC introduced a track for Question-Answering (QA) systems [18, 19, 28].

Even though this is considered as the start of modern QA research, QA systems were implemented as early as the 1960's and 1970's. However, there is no clear line of evolution from those legacy systems to the ones used today.

One of the early QA systems was BASEBALL, a program designed to answer English-written questions about baseball games for the American League, for one year [13]. Questions were read from punched cards, which was a typical medium used for data entry at the time. The words were looked up in a dictionary before the program, through some processing, determined the requested information. BASEBALL used list structures to represent information, and could for example answer questions like "Where did the Red Sox play on July 7?".

LUNAR was one of several question-answering systems developed by the researcher William Aaron Woods. The focus of LUNAR was to give answers about chemical analysis of moon rocks, collected from multiple Apollo moon trips [39].

Both of these systems are described as closed-domain systems because they were written for some specific, relative small domain. Because of this, they did not appeal to the general community but rather to a small group of people. What was common for these and other, similar early QA systems, is that they parsed natural language questions into some semantic form, specifying the information needed and finally converted this information into a machine-readable format. Data was usually stored in a structured database, but there was no cooperation between different designs, and each program would typically use its own schema. The processing approaches used in these systems were complex, making it difficult and expensive to extend them [23]. They did not scale well, and LUNAR was particularly vulnerable to sudden unexpected failures. The poor design in general and lack of expandability are key factors to why these systems were discarded in later years.

Shortly after the creation of the World Wide Web in the early 90's, a new modern QA system became available. Prager [23] called START (SynTactic Analysis using Reversible Transformations)[4] the next milestone in QA history. START is an online, web-based, open-domain question answering system developed by Boris Katz, and became available to the public as early as December 1993. According to the InfoLab Group[5] at the MIT Computer Science and Artificial Intelligence Laboratory, START was the first web-based QA system in the world. It was designed to answer natural language questions by parsing the questions to a knowledge base, and present the appropriate answers to the users. This differs from information retrieval systems where, given a query, top-ranked documents or links are returned instead. Today, START is able to answer questions about many different topics, like geography, science, history, entertainment and more, and it is still being further developed.

WolframAlpha[6] is another, quite popular answering engine used in modern times. What started as a platform for mathematical calculations (Wolfram Mathematica) has evolved into an open-domain knowledge engine that can provide and present factual information for many different topics. WolframAlpha is specifically well suited to understand natural language phrased, fact-based questions, and computes the answers using external sources. Just like MIT's START, answers and relevant information is presented in a direct format, avoiding a list of ranked documents [38].

The amount of Linked Data has in the past years increased as a result of the LOD project. This has also given more attention to QA systems over Linked Data. Today there are constantly new, relative small QA systems published around the Web. Which methods that works best are still a matter of discussion, and also depends on the different types of QA systems. As discussed in the beginning of this section, researchers attend conferences (TREC) to discuss and present new and improved methods. Question Answering was an active track in these conferences from 1999 to 2007. In 2015, this track was replaced by the Live QA track. The goal, or challenge of this track is described by NIST: "In

[4] http://start.csail.mit.edu
[5] http://groups.csail.mit.edu/infolab/
[6] `https://www.wolframalpha.com/`

this track systems generate answers to real questions originating from real users via a live question stream, in real time." [20]

2.2.2 Related work

The amount of available research papers on QA systems over LD has also increased in the past years, as a result of the growing interest for the field. Some of these papers present an introduction to the field including common methods, other focus on addressing and improving specific challenges, which requires a basic understanding of the concepts of both QA systems and LD in general. In the following section, a few published papers have been selected in order to demonstrate some of the techniques and approaches that are considered as state-of-the-art methods today. There are specifically two concepts addressed; what approaches to use for analysing a question and locate semantic information, and how to convert the processed data into a structured query.

Analysing the question

The first step of analysing a question often involves a linguistic analysis of the sentence, as well as detecting and mapping semantic information within the question [31]. The linguistic analysis can include a syntactic analysis (rules and principles for sentence structures in a given language) and a semantic analysis (meaningful concepts and phrases). A common tool often seen here is part-of-speech (POS) taggers, further discussed in Section 2.4. By assigning a grammatical tag (noun, verb, adjective, etc.) to each word, POS tag patterns can detect meaningful phrases which later can be mapped to a semantic item from the knowledge base, i.e. an entity or a relation. Domain-dependent phrases can be included in a manually constructed dictionary, where each phrase maps to one or more resources/predicates from the ontology/KB. For domain-independent terms, it's possible to use third-party libraries. For instance, WordNet [41] is a large lexical database of English nouns, verbs, adjectives and adverbs grouped into synonym sets (synsets). ReVerb is another program designed to automatically detect relationships from English sentences [11].

Question analysis can also include steps like Named Entity Recognition (NER) (c.f. Section 2.4), either through lexical phrase mapping as described above, or by using specific modules that's been designed specifically for recognising named entities. TagMe [7] is an example of such a tool. Though not exclusively used for detecting NEs, TagMe annotates and maps short phrases in unstructured text to Wikipedia pages.

Other features can also be extracted with question analysis, such as detecting question phrases, expected answer types, the general topic of the question and the overall structure of the sentence. These terms and more are presented in Section 2.3 and 2.4.

[7] `https://tagme.d4science.org/tagme/`

Query generation

When comparing different implementations, it's not unlikely that several approaches use many of the same methods for analysing a question, though with some variation. However, the next step that is responsible for constructing an executable query will probably be very different for each program [31]. Some approaches are based on templates, where either the template is predefined or constructed based on the linguistic analysis. Detected resources, properties and relations can then replace variables in the template. The paper [32] by Unger et al. presents a template-based approach, where a SPARQL template is constructed such that it mirrors the internal structure of the question, before being instantiated with entities and predicates. LODQA [8] is another example of a similar approach. Template-based approaches could work well both for domain-dependent and -independent systems.

Ontology-specific systems on the other hand requires a specific ontology, i.e. a limited domain. These systems will generally interpret user questions while maintaining a high focus on the ontology. For instance, the ontology is used for resolving any ambiguities. An example of such a system is Pythia by Christina Unger and Philipp Cimiano [30]. Pythia relies on an ontology-lexica, and "compositionally constructs meaning representations using a vocabulary aligned to the vocabulary of a given ontology" [30]. By doing so it allows for deep linguistic analysis and can handle complex questions, however the drawback is maintaining the ontology-lexica. Pythia can be complex to understand and difficult to summarise in a short paragraph, but presents an interesting approach for dealing with complex questions, and readers are encouraged to read the full paper.

Another method is to create a graph of the detected resources and relations, and then exploring the graph in order to find the connection between them by utilising the knowledge base. However, searching through all the possible connections may be an exhaustive task, and systems of these kinds would typically need a limit for how deep into the graph the search should go [32].

2.2.3 Question Classes

A question can have many forms and can ask for different things. It's obvious a difference between a calculation question such as "What is the square root of 16?" and a definition question like "What is Norway?".

For a general QA system, there is no formal definition of what a question is or how it should be written [23]. But when looking at the answers for these questions, what's typical is that these answers are not *generated* from the information, they are *extracted* from some database or knowledge base. This is a key concept to QA systems today. Answers are extracted from either a knowledge base, a database or even a large corpus of text, depending on the type of QA system. And all of these answers, or text passages from where the answer is extracted, have been typed in and checked by somebody.

[8]http://www.lodqa.org/docs/intro/

In order to work with different kinds of questions, regarding both the implementation and evaluation, it's a good idea to categorise these questions into classes.

Factoid questions

A factoid is similar to a fact, but may not necessarily be equal to one. It could be an invented fact believed to be true because it appears in a newspaper or on a web page [16], it could be a false statement presented as a fact [10] or it can actually be a real fact.

There is no clear definition of what a factoid question is, but in general, factoid questions are those types of questions that seek a simple fact or even a relationship. Examples could be "How high is Mount Everest" or "Who is Barack Obama married to?". The answer is often just a few words or a short sentence. These types of questions were quite popular in the beginning of TREC-QA tracks, and TREC "defined" a factoid question with regards to the returned answer; It should return just a simple string of characters, and it should differ from list and definition questions (described subsequently) [23].

How and *why* types of questions often look for definitions and explanations, and would not be classified as factoid questions. Examples are "How do I make a pancake?" and "Why are German drivers so successful in Formula 1?". Clearly, there are no short, simple answers. However, "how" and "why" questions do not necessarily have to be excluded from factoid questions, like the example above; "How high is Mount Everest?" - which indeed has a simple answer.

List questions

List questions are those types of questions that do not seek one specific answer, but a set of multiple, factoid answers. The set might also only contain one answer or none at all.

Who drove for McLaren in the 2011 Australian Grand Prix?

List the Italian drivers that participated in the 2012 season

When a QA system answers list questions, it's usually important to include all possible answers that might be correct. However, some lists may be excessively long and might need to be abbreviated using some confidence threshold [23].

Definition questions

Some questions asks for answers that cannot be easily converted into a simple sentence. Questions in the form of *Who is someone?* or *What does something do?* might need a collection of facts in order to construct an answer. In the overview of the 2006 TREC-QA track [8], a *definition question* is defined as a

question that asks for interesting information about a particular person or thing, and also requires systems to find information from multiple documents. Note that the "answer" here is not a specific answer, but some amount of information.

One of the challenges with definitions is that they are influenced by the people who wrote the fundamental information. Answering definition questions is also a challenge because of the amount of information that could be available. A system that aims to answer these questions needs to have some control of how much information to return. For example, if the question is "Who is Michael Schumacher?", is it enough to explain that he is a former Formula One racing driver? How much of the various information about him should be returned?

These kinds of questions were dropped by TREC in 2002, because they did not fit into the factoid model. They were later reintroduced under the category "Other", which was a category interpreted as "Tell me other interesting things about this target I don't know enough to ask directly" [8], where the target (people, organisation, things, etc.) was given.

Relationship questions

Relationship questions were introduced in the 2005 TREC, and should not be confused with the "factoid relationship questions". As described above, a factoid question can ask for a relationship between two entities if that relationship can be presented as a fact or factoid. The difference is that a relationship in this context, as its own category, refers to the ability of an entity to influence or affect another entity [23, 8], with the purpose of finding some "evidence", which can include both the means and motivation to influence something [17].

Other types of questions

In addition to the most popular categories described above, some systems also include other classes, such as

- Yes-no questions
- Opinions
- Cause and effect
- ...

These classes are not necessarily implemented equally across different QA systems, and it's also possible to use own made classes.

2.3 Overview of QA terminology

This section presents some terminology commonly used in QA systems, such as question and answer types. Some of the examples presented in this section are closely related to the domain of Formula 1, which is the topic of the implementation presented in Chapter 3.

2.3.1 Question Types

Question types are similar to the ones described in Section 2.2.3. It is a way of categorising questions into types or classes of questions with regards to the expected results. These classes, as already covered, could be list questions, factoid questions, definition questions or other types. It's also possible to create own types of questions when developing a QA system, but many of the types already described have been a standard for different kinds of evaluations.

2.3.2 Question Phrases

These phrases are the part of the question that gives an indication of what the question is looking for. They are typically *wh-words*, like "who", "what", "where", "why", "when", "which" and "how", along with words they modify. They may stand alone in a sentence, like "Who won the Italian Grand Prix in 2014?", but quite often they also stand together with an adjective, a noun or an adverb; "How fast...", "Which teams..." etc.

List questions can be expressed by these question phrases as well, but it is not uncommon to see list questions of the type "Name a ..." or "List the ...".

Question phrases are also closely connected to *answer types*.

2.3.3 Answer Types

Answers can be categorised into different types or classes, and the *answer type* in this context refers to the class the answer belongs to. These classes may be derived from the knowledge base. For example, a driver is a type of **person** and a country is a type of **place**. These classes can often be determined by the question phrase;

Who - Person or organisation
Where - Place
When - Date or time
Which driver - driver - person

There are many options for mapping each question phrase to an object class. From the question phrase *Which driver* it's possible to extract *driver*, and then map it to a type of Formula One Driver, which again is a type of person. What types to use really depends on the structure and hierarchy used in the knowledge base, i.e. the ontology, in addition to personal preferences.

The answer type may have some impact on the final query, or in the ranking procedure of the results, but not necessarily. Consider the two, relative similar questions below.

1. *Which team won the Chinese Grand Prix in 2012?*
2. *Who won the Chinese Grand Prix in 2012?*

The question phrase in the first question clearly indicates that the answer type should be a team. However, for the second question there are two possible correct answers. Both the winning driver and the winning team will answer the question correctly. The question phrase "Who" does not indicate if the answer type should be a person or a team. For this challenge there are some methods that can be implemented:

1. Return both the driver and the team.

2. Use some weights for each candidate, where the candidate with the highest weight is returned. These weights could be determined by collecting statistics from previous queries. For example; "There is 78% probability that the question asks for the winning driver".

3. When in doubt and answers of different types are present, ask the user to clarify the answer type.

2.3.4 Question Focus and Topic

Sometimes closely related to answer types are *question focus*. The focus is generally a noun or a noun-phrase that points to the property or entity that the question seeks [23].

The *question topic* refers to the object the question is generally about. Consider the following examples:

Example 1: *What is the population of Japan?*

Example 2: *How many champion titles does Michael Schumacher have?*

The question focus in example 1 is the property population. The topic is Japan. The topic of example 2 is Michael Schumacher, because the question is generally about him. The sought property of this question is "champion titles", i.e. the question focus.

2.3.5 Authority Lists

An authority list refers to a collection of instances of known answer types, and is often used to test the class-membership of a term. Examples of such lists could be weekdays, planets, countries, US presidents, etc. This allows a system to check that the term "Norway" is a type of country (assuming that Norway is included in the set of countries).

Good authority lists should be relative small and preferably limited. Animals, plants and colours can be good collections, but it's often difficult to construct a complete set. Lists of numeric quantities such as dates, population and numbers are unnecessary and inappropriate [23].

2.4 Natural Language Processing

One of the most important tasks of a QA system is to analyse the question, a text written in natural language. In the field of computer science, this intersection of human language and computer understanding is called Natural Language Processing (NLP). It consists of different methods, techniques and approaches that's used to analyse human language, with the ultimate goal of understanding the meaning behind the text or question. This includes both analysing words and their grammatical meaning, as well as the structure of the sentence.

NLP is used for many different kinds of applications, and not only for question answering systems. Some examples are grammar correction in text processing programs, translating text into other languages, speech recognition and more [2]. Some of the popular approaches used in NLP for question answering systems are presented below.

Part-of-speech tagging

Part-of-speech (POS) tagging refers to the concept of classifying each word in a sentence to a specific tag that describes the word's grammatical meaning in the sentence, like verbs, nouns, adjectives, adverbs etc. This process is also referred to as just 'tagging'. The set of predefined tags used for a specific program is known as a tagset [3]. Typically, a programmer would import a tagset from a library, for example the NLTK POS tagger [3] or the Stanford POS tagger [14], though it's also possible to modify these with custom tags or even create a tagset from scratch.

The following example demonstrates a tagged sentence, retrieved from [2]. The tags used below are typical for most POS taggers: NN (noun), VB (verb), JJ (adjective), RB (adverb).

Semantic/JJ search/NN is/VB just/RB great/JJ.

A related method used for natural language analysis is *chunking*. The process of chunking refers to taking sequential words that belong together based on their POS tags, join them as one object and tag the object with a "chunk tag". Chunks can be seen as non-overlapping trees, where the leaves of the chunk-tag are either new trees or individual words. Also note that it's not necessary to chunk all the words in a sentence [2].

Likewise to POS taggers, chunking algorithms are available through libraries such as NLTK, but could and often should be optimised by the use of e.g. regular expressions to detect domain-specific patterns. For example, consider the phrase "2011 Australian Grand Prix", which refers to the specific Formula One Grand Prix hosted in Australia in 2011. A common tag- and chunk process could yield the following result:

2011/CD Australian/JJ (NP Grand/NNP Prix/NNP)

The result means that '2011' is tagged as a cardinal digit (CD), Australian as an adjective (JJ), the terms 'Grand' and 'Prix' are both tagged as proper singular nouns (NNP), but chunked together to a noun phrase (NP). However, given the domain of Formula 1, the desired result would be to chunk the whole phrase into one entity, for example tagged with 'GP' (Grand Prix). This could be fixed with a regular expression that looks for the POS tag-pattern *<CD><JJ><NP>* or *<CD><JJ><NNP><NNP>*, and then compare the GP-tagged phrase to a list of Grand Prix entities from the knowledge base.

The example above is not only demonstrating chunking, but it's also an introduction to what chunking is often used for; Chunking text can be a good first-step towards both Named Entity Recognition and sentence parsing [2], which will be described below.

Named Entity Recognition and Disambiguation

Given a natural language question, one of the tasks that needs to be solved before looking up an answer is to detect named entities (NEs), like places, people or organisations. This is called Named Entity Recognition (NER). More precisely, it means to detect any text-phrase (sequence of words) in the question, that may refer to an entity from the knowledge base. Actually, there is no requirement of a knowledge base, because the task of NER refers to recognising any phrases that might refer to an entity, for example a set of proper nouns. However, using a knowledge base in addition to a general NER function can improve the identifying process, especially if there are uncommon named entities present in the knowledge base, like the Grand Prix example above. The knowledge base also becomes important in the next step where potential entities are mapped against it.

After identifying phrases that might be entities, the next step called Named Entity Disambiguation (NED) takes care of determining if the entity exists in the database, and which entity it refers to. As discussed in Section 2.1, a word can have many different meanings depending on the context it's used in, and the task of a disambiguation process is to determine which specific meaning a word has, or which specific entity that it refers to in the context. For example, in the sentence "Armstrong travelled to the Moon" [9], it should be clear that Armstrong refers to the astronaut Neil Armstrong and not the cyclist Lance Armstrong, and that the Moon refers to the Earth's moon, and not another moon in the solar system. In this example, the overall goal of NED is to detect the link between "Armstrong" and "the Moon", so that it can with some confidence return the correct entities. (For this purpose, NED is also known as *Named Entity Linking* [2].)

To summarise NER and NED, consider the following question:

Did Schumacher win the 2005 USA Grand Prix?

[9] Example retrieved from [2]

The task of NER is to identify 'Schumacher' and '2005 USA Grand Prix' as possible entities. The task of NED is to map them to the applicable entities in the knowledge base. The term 'Schumacher' alone can be difficult to map correctly, because it can refer to both Michael and his brother Ralf, who are both Formula 1 drivers. However, linking the information with the 2005 USA Grand Prix entity, it should be possible to detect that Michael Schumacher was the driver who won that race.

Sentence Parsing

Sentence parsing is a way of analysing the sentence, or question, with the goal of identifying the grammatical structure. This is similar to the POS tagger, but the focus here is on the sentence as a whole, and not the individual words. Today there are two common procedures used for sentence parsing, *the constituent parse* and the *dependency parse*, which both can be viewed as a tree [2].

```
S(NP((Semantic) (search)) VP(VB(is) ADJP((just) (great))))
```

Listing 2.2: Constituent sentence parsing

A constituent parse of a question consists of recursively splitting the sentence into phrases until the level of words or chunks is reached. The root node is called S, demonstrated in the example above.

```
Semantic <- search <- is -> great -> just
```

Listing 2.3: Dependency sentence parsing

With dependency parsing, all the words point to exactly one other word in the sentence which they depend on. This can be viewed as a tree, where the root node of the tree is the main word in the sentence. Listing 2.3 demonstrates an example. Both examples above are extracted from [2].

Chapter 3

Implementing a QA System

A first version implementation of a domain-specific QA system over Linked Data is presented along with a prototype presentation. The QA system covers the domain of Formula 1 statistics, and the knowledge base includes some general information about drivers, teams and Grand Prix - in addition to race statistics for races. This chapter also presents the ontology used for this domain. More detailed information will be presented in the corresponding sections.

The implementation will only include and deal with factoid and list types of questions. Other types will not be considered.

3.1 Ontology

The goal with this implementation is to be able to answer natural language questions for the domain of Formula One statistics. The domain covers knowledge about statistics for drivers, teams, races (or Grand Prix) and seasons. This section introduces the ontology, i.e. the hierarchy and overview of classes and their properties.

The knowledge base, which will be covered in the next section, is constructed by extracting statistical information from Wikipedia. Hence, much of the vocabulary and namespaces can be reused from DBpedia. To recap, DBpedia is a knowledge base that provides a structural, semantic representation of Wikipedia data. However, DBpedia was not able to provide all the appropriate classes, properties and relations required for this domain. To compensate for this, classes and properties of a private repository has been implemented.

The base URL for this repository is `http://f1answers.fake`. In addition to this and DBpedia, this ontology also includes some common namespaces from w3.org. These are presented in Table 3.1 below.

Prefix	URI
dbo	`http://DBpedia.org/ontology/`
dbp	`http://DBpedia.org/property/`
dbr	`http://DBpedia.org/resources/`
f1o	`http://f1answers.fake/ontology/`
f1p	`http://f1answers.fake/property/`
f1r	`http://f1answers.fake/resources/`
owl	`http://www.w3.org/2002/07/owl#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs	`http://www.w3.org/2000/01/rdf-schema#`
xsd	`http://www.w3.org/2001/XMLSchema#`

Table 3.1: Table of namespaces and prefixes used in the ontology

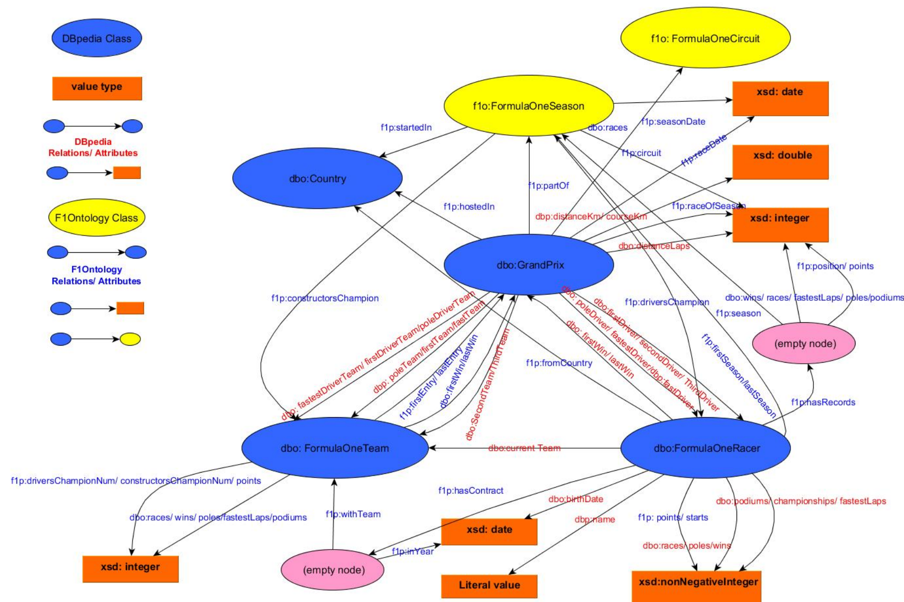

Figure 3.1: Formula 1 ontology used for the QA system

When developing an ontology, it's often useful to think about what kind

of information it should provide in the end. Without a sense of what kind of information to provide, it's difficult to know where to start. For the domain of Formula 1, it's firstly interesting to see statistical information about drivers, teams and races, in addition to factoids about them. In order to get a full overview of the ontology, it's better to examine the visual representation rather than with pure text. Figure 3.1 presents the first version of the ontology used in this QA system. The most important classes and properties will be further described below, the other relations and properties not covered can be found in the figure, and readers should find it simple to understand. In the figure, classes are represented by circles. Properties are defined by labelled, directed arrows, and the property values are either a type of another class (the arrow points to another circle) or a literal value (the arrow points to an orange rectangle).

Formula 1 (F1) is the highest category or class of single-seat auto racing, and is a popular and globally well-known sport. A season in F1 consists of multiple races called Grand Prix (GP), and these races are hosted all around the world [12]. `GrandPrix` is the only class that links to all the other classes, which makes it an important and central node in this ontology. In Figure 3.1, classes from the DBpedia ontology are marked with a blue background. `GrandPrix` has properties that links it to other DBpedia classes, but also to those that are defined for this ontology (f1o). These are recognised by the yellow background. Some of the properties of `GrandPrix` are the following:

`hostedIn` describes which country a GP was hosted in. The proper name for a GP is usually in the format of *<Year> <Country> Grand Prix*, for example *2011 Australian Grand Prix*. In these cases it's an easy task to understand which country that hosted the GP. In other cases, such as the *European Grand Prix*, it's not so clear. By including the property `hostedIn`, it's possible to retrieve all the GP that were hosted in a specific country.

In each GP there are a number of different teams competing, and therefor it's logical to include the roles between races and teams. In this ontology there are properties in both directions between the two classes `GrandPrix` and `FormulaOneTeam`. `GrandPrix` has properties such as `firstDriverTeam`, `poleDriverTeam` and `fastestDriverTeam`. These are all properties found in the DBpedia ontology. The *first driver team* represents the team that won the race, and the *fastest driver team* represents the team that had the fastest lap time. A pole position in Formula One is a special position at the start of the race, often given to the driver who had the best qualifying time, who in turn is referred to as the *pole driver*. This position is typically located at the front in the starting position grid. A GP can also be the object of a property for `FormulaOneTeam`. `firstWin` and `lastWin` are properties that provide information about what GP a specific team won for the first or last time.

The relation between a GP and a `FormulaOneRacer` is similar to the relation to teams. Equally important as to knowing which team that won a race, is it to know which driver who won a race. In addition, the ontology includes properties describing which drivers who came in second or third place, and who the pole driver was. These are also represented in the DBpedia ontology.

Because a GP is hosted within a season, in addition to the fact that there

are interesting things to ask about a given season, it's also naturally to include the relation between a season and a GP. The `partOf` property defines that a GP is *a part of* a F1 Season. This property along with the class `FormulaOneSeason` did not have a direct relation in the DBpedia ontology, and was hence created specifically for this ontology.

The property `circuit` defines the relation between a GP and a circuit, the course that hosted the race.

Because statements are represented by triples in the RDF framework, there are some relations that are difficult (and impossible) to express. For example, defining how many points a driver got in a specific race cannot be expressed by one statement in this ontology. It is theoretical possible to define it using only one statement, but this would require one property for each Grand Prix. For instance, if the race was the Australian Grand Prix in 2011, a property called pointsIn2011AustralianGrandPrix would have solved the problem. Clearly, this makes no sense and does not work in the long run.

This kind of problem is similar to the "many-to-many relation" problem often found when constructing a relational database. A many-to-many relation would typically be split up using a third node in between the other two nodes, converting the many-to-many relation into two one-to-many relations. In RDF ontology, this third node is called a *blank node*, or *empty node*. A blank node can describe complex relations or attributes, and represents a resource or a concept. The node does not have it's own URI [5].

The F1 ontology presented here includes two blank nodes. One of them joins a F1 Racer with a F1 Team. This was necessary in order to describe the historical relations a driver had with a team. Over time, a F1 racer will typically drive for and represent many different teams. In other words, a driver has a contract with a team, where the contract is valid for one or more seasons. This is represented in the ontology by giving the class `FormulaOneRacer` a property called `hasContract`, which points to an empty node. This empty node again connects to `FormulaOneTeam`. In addition, the empty node has a property called `inYear`, referring to which year (or season) the contract was valid for. In summary, this describes a contract between a driver and a team for a given year/season. Note that the blank node could have been omitted if it wasn't necessary to describe for which seasons a contract was valid. If it only was interesting to see former teams of a driver without concerning about the timeline, then these contracts could have been expressed in simple triple-statements. For example, a F1 racer will only have one "current team". Therefor it's possible to have a direct relation, through the property `currentTeam`, between a racer and a team as illustrated in the figure.

The second empty node connects a F1 racer to a `FormulaOneSeason`, through the the properties `hasRecords` and `season`. More specifically, the empty node includes statistical information such as number of wins, points, podium positions (top three) and the overall position for a given season, and more. With this knowledge it's possible to answer questions like "Who got the fourth place in the 2011 season?", "Which driver had the most top-three positions in 2012?" etc.

3.2 Knowledge Base & Lexicons

3.2.1 Data collection

Collecting data and building the knowledge base (KB) is not a contribution of this paper, but a short summary of what's been done is presented below.

First of all, data has been collected using Wikipedia as the main source. For statistics and facts, Wikipedia has shown to be a good source and presents information in well-structured tables. One of the benefits of using Wikipedia is DBPedia, the KB of Wikipedia data. Reusing existing vocabulary can save time as well as increase the quality of the ontology, compared to making one from scratch. Another advantage of Wikipedia is that it has a similar structure for every Grand Prix, driver and season (i.e. entities) making it possible to automatically collect data with code-scripts.

3.2.2 Tools

The KB itself was instantiated with the tool GraphDB[1] by Ontotext. GraphDB is available as a free download, and has shown to be a great tool for developing a KB. In addition to a hosting server (the triplestore), it comes with a user-friendly workbench, allowing easy manipulation of data. The tool includes a SPARQL interface for running queries, as well as automatically-created graphs for data statistics, e.g. how instances are distributed on classes and which kind of classes that has the most in- and out-links to other classes.

3.2.3 Lexicons

As shown in Figure 1.1 in the introduction, the implementation includes a set of lexicons that are constructed offline as part of system maintenance. One lexicon stores a list of all resources available in the KB, and is used for mapping named entities. The other lexicon stores a list of ontology classes in addition to semantic relations from the ontology, i.e. predicates, along with a list of natural language phrases for each predicate. This lexicon is used for detecting and mapping answer types and semantic phrases, i.e. converting natural language phrases to semantic items.

The lexicons are constructed with a script written in Python 3.0 (see attachments). The script executes a SPARQL query that retrieves all the triples from the KB, and separates the subjects, predicates and objects. Subjects and objects are used to build a lexicon of resources, and the predicates are used for building a lexicon of phrases related to ontology vocabulary.

Resource-lexicon

The lexicon of resources stores the URI, the common name or label and finally a list of all the entity types it can relate to. Because subjects and ob-

[1] `https://ontotext.com/products/graphdb/`

jects in the RDF triples can point to the same resource, these are merged into one set of distinct objects. A regular expression is used to pull out objects that points to a resource, by checking for the term "/resource/" in the object's URI, e.g. `http://dbpedia.org/resource/2015_Belgian_Grand_Prix`. For converting this URI into a label, another regular expression extracts the string after "/resource/", and replaces the underscores with whitespaces. It's also possible to run a query that collects the object's label from the KB, but not all resources have the same property for labels and names in the KB. Some resources don't have a property for a label at all.

The lexicon also stores a list of ontology classes for each resource. This list is constructed by executing a SPARQL query with the RDF triple pattern <resource> rdf:type ?type. This query is first run towards the private KB, i.e. *f1answers.fake*. If there are no answers/types found in this KB, it then tries to run it on the DBPedia KB. When completed, the list of resources is written to a text file (included in attachments). A short excerpt of this file is shown in Figure 3.2 (full URIs are replaced with prefixes for demonstration purposes only).

```
dbr:2009_Spanish_Grand_Prix     2009 Spanish Grand Prix [dbo:GrandPrix dbo:Event ...]
dbr:2009_Turkish_Grand_Prix     2009 Turkish Grand Prix [dbo:GrandPrix dbo:Event ...]
dbr:2010_Abu_Dhabi_Grand_Prix   2010 Abu Dhabi Grand Prix       [dbo:GrandPrix dbo:Event ...]
dbr:2010_Australian_Grand_Prix  2010 Australian Grand Prix      [dbo:GrandPrix dbo:Event ...]
dbr:2010_Bahrain_Grand_Prix     2010 Bahrain Grand Prix [dbo:GrandPrix dbo:Event ...]
dbr:2010_Belgian_Grand_Prix     2010 Belgian Grand Prix [dbo:GrandPrix dbo:Event ...]
dbr:Scuderia_Ferrari    Scuderia Ferrari        [dbo:FormulaOneTeam dbo:Agent dbo:SportsTeam ...]
dbr:Scuderia_Milano     Scuderia Milano []
dbr:Sebastian_Vettel    Sebastian Vettel        [dbo:FormulaOneRacer dbo:RacingDriver ...]
dbr:Sergio_Mantovani    Sergio Mantovani        [dbo:FormulaOneRacer dbo:RacingDriver ...]
dbr:Sergio_Pérez        Sergio Pérez    [dbo:FormulaOneRacer dbo:RacingDriver ...]
dbr:South_Africa        South Africa    [dbo:Country]
dbr:South_Korea South Korea     [dbo:Country]
dbr:Spain       Spain   [dbo:Country]
f1r:2011_Formula_One_Season     2011 Formula One Season [f1o:FormulaOneSeason]
f1r:2012_Formula_One_Season     2012 Formula One Season [f1o:FormulaOneSeason]
f1r:2013_Formula_One_Season     2013 Formula One Season [f1o:FormulaOneSeason]
f1r:Circuit_de_Barcelona-Catalunya      Circuit de Barcelona-Catalunya  [f1o:FormulaOneCircuit]
f1r:Circuit_de_Catalunya        Circuit de Catalunya    [f1o:FormulaOneCircuit]
f1r:Circuit_de_Monaco   Circuit de Monaco       [f1o:FormulaOneCircuit]
f1r:Sebring_Raceway     Sebring Raceway [f1o:FormulaOneCircuit]
f1r:Sepang_International_Circuit        Sepang International Circuit    [f1o:FormulaOneCircuit]
f1r:Shanghai_International_Circuit      Shanghai International Circuit  [f1o:FormulaOneCircuit]
```

Figure 3.2: Excerpt of resource-lexicon

The resource-lexicon is automatically created by the script and requires no manual intervention. This means that it's possible to execute the script on a regular basis in order to stay updated.

Phrase-lexicon

The lexicon of phrases is based on the set of ontology predicates. The script collects all the distinct predicates and store these in a separate file (included

in attachments). However, with the current implementation this lexicon needs manual intervention in order to include natural language phrases for each entry. Chapter 5 discusses how this can be resolved in the future. An extraction of the current phrase-lexicon is shown in Figure 3.3 .

```
dbo:secondDriver        {'second', 'second place', '2nd place', '2nd', 'runner up',
                        'runnerup', 'second driver'}
dbo:secondTeam          {'second', 'second place', '2nd place', '2nd', 'runner up team',
                        'second place team', 'second team'}
dbo:thirdDriver         {'third', 'third place', '3rd place', '3rd', 'third driver'}
dbo:thirdTeam           {'third', 'third place', '3rd place', '3rd', 'third place team',
                        'third team'}
dbo:wins                {'wins', 'number of wins', 'count of wins'}
dbp:courseKm            {'course', 'course length', 'course distance', 'distance'}
dbp:distanceKm          {'race distance', 'distance', 'race length'}
dbp:name                {}
dbp:poleTeam            {'pole driver team', 'pole team', 'pole'}
f1o:FormulaOneCircuit   {'circuit', 'course', 'circuits', 'courses'}
f1o:FormulaOneSeason    {'season', 'year', 'years', 'f1 seasons', 'f1 season', 'seasons',
                        'formula one season', 'formula one seasons'}
f1p:circuit             {'circuit'}
f1p:constructorsChampion        {'champion', 'constructor champion', 'champion team', 'winner'}
```

Figure 3.3: Excerpt of phrase-lexicon

3.3 Approach

The following subsections describe the approach and the components used for the implementation of this Question Answering system, from converting a natural language question into one or more SPARQL queries and retrieving an answer. The program has been implemented using Python 3.0 programming language (see attachments for source code). An overview of the QA module pipeline is shown in Figure 3.4. A full example of all the steps in this pipeline is demonstrated in the next section.

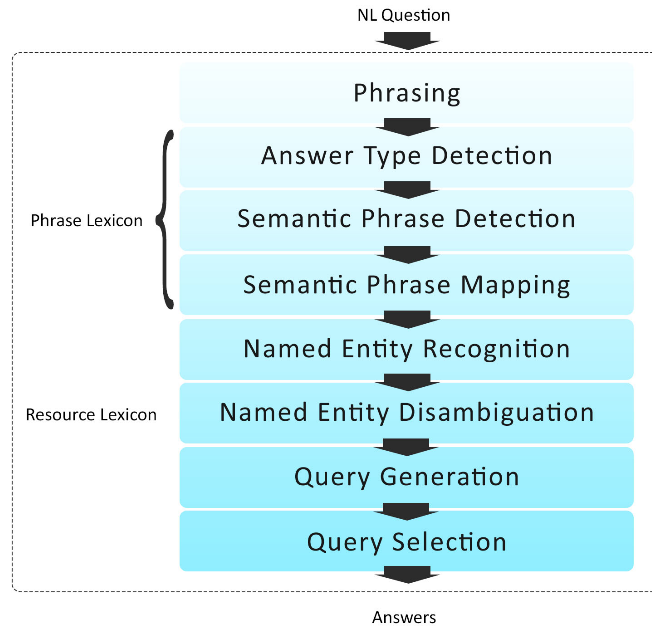

Figure 3.4: Pipeline of Question Processing Module

3.3.1 Phrasing

The first step in the pipeline of this QA system is called phrasing, inspired by the approach done by Yahya [43]. This refers to splitting up the text into phrases that potentially can be mapped to a semantic resource or relation.

The question-text is first tokenized using the NLTK word tokenizer [21], which splits the text into multiple sub-strings (tokens) consisting of the words and punctuation marks in the text. Splitting up the text like this is intuitively enough done because it's much easier to analyse and work with a text word by word, the same way children analyse sentences in elementary school. In addition, tokenization is needed in order to mark each word in the sentence with its grammatical meaning, referred to as Part-of-speech (POS) tagging as described in Section 2.4. The POS tagger used in this implementation is from NLTK as well, but other options are also available, such as the *Stanford Log-linear POS Tagger* [2].

Once the text has been preprocessed by tokenization and tagging, the tagged text can be chunked into phrases. Section 2.4 described how a text can be chunked using the NLTK Named Entity chunking function, but this method alone focus on proper nouns, and not other types of phrases. However, it's possible to split a text into chunks by using POS patterns created by regular expressions. This makes it possible not only to chunk possible entities, but also phrases of any

[2]https://nlp.stanford.edu/software/tagger.shtml

pattern desired. When considering domain-specific applications such as Formula One, third party libraries might be insufficient for recognising special patterns, that may only occur within the specific domain. This implementation excludes the NE chunking from NLTK (or any other library), with the purpose that the POS pattern chunking recognises potentially named entities, and presents the following patterns used for chunking a text into phrases.

<DT><RB.?> - A determiner (the) followed by an adverb. Examples are "the most", "the highest", "the longest", "the least", etc.

<WP> - Wh-pronouns, such as "Who" and "What".

<WRB><JJ.?> - Wh-adverb followed by an adjective. An example may be "How many", where *How* is tagged as wh-adverb and *many* is tagged as an adjective. The purpose of this pattern is to recognise questions of the type "How many", "How far", "How much", "How long" etc. The question phrases when, where and why are also among wh-adverbs, but are usually not followed by an adjective.

<JJ.?><NN.*>*<IN><CD> - This pattern is constructed by an adjective, followed by any number of nouns, followed by the phrase "in <CD>", where CD is a number. This pattern is mostly for recognising phrases such as "Italian Grand Prix in 2014", which could be mapped to an entity later on. This is an important pattern used for NER for types of Grand Prix.

(<CD>|<JJ.?>)*<NN.*>* - This pattern is more relaxed than the previous pattern, but is also used for, amongst other, recognising Grand Prix entities. Examples are "2014 Australian Grand Prix" and "2015 Season". The pattern is constructed by any repetitions of adjectives and/or numbers, followed by any number of nouns. Not only will it recognise Grand Prix entities, but it also takes care of any noun phrases that are described by an adjective, such as "big, red monster truck".

<.*> - This final pattern will match any tag, meaning that all of the remaining words will be chunked into their own phrases.

The order these patterns are presented in also affects the chunking process. If a text matches multiple patterns, it will only be chunked using the first pattern it matched. This can become a problem if a substring that were supposed to be chunked by a specific pattern down the list, has a partial matching for a pattern higher up the list. For example, if the final POS pattern <.*>, which matches any tag, were listed at the top of the POS-pattern list, then none of the other patterns would ever contribute to chunk a text. All of the words would have been chunked as its own phrase.

To sum up, the component called phrasing consists of splitting the words into tokens, apply POS tags and chunk the words into phrases. The idea is that

these phrases could later be mapped to either resources (or Named Entities) or semantic phrases (i.e. predicates from the ontology).

3.3.2 Determining answer type

The Answer Type (AT) module presented here uses multiple functions that first of all try to detect the Question Phrase (QP), and secondly looks for AT candidates.

The first part consists of searching through the words and phrases in the question, looking for one of the QP in Table 3.2. If a QP from the question is found in the table, the module will look up the corresponding AT. These AT are for internal use only and used for further processing.

Question phrase	Internal Answer Type
name	LIST
list	LIST
who	Person, Organisation
where	Location
what	Something
when	Date
which	which
how	how
is	NA
are	NA
was	NA
were	NA
why	NA

Table 3.2: Look-up table of questions phrases, mapped to internal answer types

For example, if the result of the first step is *Person, Organisation*, then these types are further mapped to ontology classes, e.g. `FormulaOneRacer` and `FormulaOneTeam`. On the other hand, if the results are "list", "how", "something" or "which", custom functions takes care of detecting the AT. The reason for implementing special functions for these phrases is because it's not possible, or at least not easy, to detect what kind of answer the question seeks given only the QP. A "list" can be a list of drivers, a list of teams, a list of races and so on. The word "what" may seek something or a property of something, but further

analysis are required to find the AT.

These functions involve looking for words that the QP describes, and map these words to either classes from the ontology, or to non-classes defined in a local look-up table. If the question asks for "how many points", "at what position", "how far" or other types of properties that can be mapped to a number, the result of the AT module is NUMBER. "What racer" or "which driver" will both be mapped to the class `FormulaOneRacer`.

QP that maps to "NA" are types of questions that are not expected to be covered in this implementation. These types are not processed any further in the program, but by including them it's possible to create a specific response to the user that these types of questions are not covered in this domain.

If there is no match in the first step, meaning that no QP was found in the question, then the module failed to recognise an answer type. This doesn't prevent the system to proceed with the processing, but it could prevent it from detecting the right types of answers and hence returning multiple answers that aren't correct. However, it's expected that users ask questions using one of the supported phrases.

3.3.3 Semantic Phrase Detection and Mapping

As covered in Section 3.2.3, there is an offline part of the program that constructs lexicons for both resources and phrases. The lexicon of phrases consists of ontology predicates and classes along with common, natural language phrases that could relate to these. The first step of the pipeline, called phrasing, takes care of splitting the question into a set of phrases. The next step in the pipeline (after finding AT candidates) is called Semantic Phrase Detection. Because the phrases are already "determined", this step consists of checking each phrase up against the lexicon of phrases. If there are at least one phrase that can be mapped to a semantic item, then the phrase is considered as a potential semantic phrase.

The next step, called Semantic Phrase Mapping, takes care of determining what semantic item the phrase could or should be mapped to. Recall that a natural language phrase could easily be mapped to a number of different semantic phrases. The URI for each semantic item that is a potential candidate for the phrase is then added to a list, which act as the basis of potential predicates later used in query generation.

3.3.4 Named Entities

There are no third party libraries used for recognising Named Entities (NE). The application depends on the POS patterns for chunking phrases, with the idea that each NE is chunked as a single phrase. This corresponds to the Named Entity Recognition (NER) part. The only additional processing applied to NER, is checking if there is at least one noun in the phrase, using the POS tags. If there are no nouns, then the NE candidate is discarded. For the results on how good this approach is, see evaluation results in Section 4.

Named Entity Disambiguation (NED) is a two-part process. The first part consists of constructing a dictionary of resources extracted from the KB. As described in Section 3.2.3, this is done "offline" and manually as part of system maintenance. The second part, the actual mapping of phrases to entities, is done by a similarity function that retrieves the most similar entity/entities from the lexicon. This is of course carried out every time a question is being processed.

```
def similarity(a, b):
    a = a.lower()
    b = b.lower()
    try:
        a = " ".join(x for x in sorted(a.split()))
        b = " ".join(x for x in sorted(b.split()))
    except:
        pass

    diffl = SequenceMatcher(None, a, b).ratio()
    sor = 1 - distance.sorensen(a, b)
    jac = 1 - distance.jaccard(a, b)

    # Return the mean (average) of the three methods
    sims = np.array([diffl, jac, sor])
    score = np.mean(sims)

    return score
```

Listing 3.1: Similarity function

Listing 3.1 presents the source code of the similarity function. Note that this code only includes the part of string similarity, and not the part of checking the lexicon.

The similarity function compares two labels and yields an overall score. The first string is the phrase holding the possible entity, the second phrase is a label from the resource lexicon. A loop takes care of checking all the labels in the lexicon. The similarity function uses three different kinds of third party similarity/distance functions: The Jaccard index[3], the Sørensen-Dice coefficient[4] and a sequence-matcher from the difflib library [5]. Each of these calculates a similarity score based on their algorithms, and in the end the function returns the average of these scores. The Sørensen-Dice and Jaccard indices are measures of distance (from the distance package), and not directly similarity functions. Subtracting the score from 1 converts the distance into a similarity.

The result of this comparison is a set of entity-candidates along with a ratio number which indicates how good the match was. This ratio is in the range of

[3] https://en.wikipedia.org/wiki/Jaccard_index

[4] https://en.wikipedia.org/wiki/Sorensen-Dice_coefficient

[5] http://lxml.de/3.1/api/private/difflib.SequenceMatcher-class.html

0 to 1, where 1 indicates a perfect match. Additionally, it's possible to set a threshold for how high a similarity score must be, in order to include the result in a set of candidates. The reason for using a scoring mechanism as opposed to simply checking if the two strings are equal, is because it's unlikely that a user will enter the specific label of an entity, which means it is not enough to check for 100% matches. For example, a Grand Prix entity might present itself in a question in the form of "Australian Grand Prix in 2015". The label for this entity in the lexicon is "2015 Australian Grand Prix". These two strings are not equal, but will however yield a high score for similarity. Challenges associated with the similarity function is presented in Chapter 5.

3.3.5 Query generation

The component Query Generation (QG) is responsible for constructing SPARQL queries, using the output of the other components as input data. This includes the semantic phrases used as possible predicates, the NEs as possible resources and the detected answer types. The QG module consists of four sub-functions (or handlers) that are used for different purposes. An initial analysis of the input (predicates, resources and answer types) determines which of the four sub-functions that should be used. One of the sub-functions is a default handler, whereas the other three are for special cases. The final step of QG is a separate function that runs the queries to the KB and collects the answers. This is referred to as Query Selection in the pipeline.

The initial analysis starts by checking all the resources for class types, and stores the distinct ones in a list. The reason for this is that in some cases, given instances of known classes, it's possible to guess with a high probability what kind of relation in the ontology the question is looking for, or in some cases *not* looking for. Consider the following example.

Assume there is question with two NEs recognised and mapped, one instance of a Grand Prix and one instance of a Formula One Racer. From the ontology model (c.f. Figure 3.1), there are a few predicates that directly links these two classes. However, if the question specifically mentions a Grand Prix and a driver, it is unlikely that the question seeks one of these "direct predicates". If it did, the question would sound like *Was Michael Schumacher the winner of the 2004 European Grand Prix?*, and would be categorised as a "Yes/No" question, not covered in the domain.

A question like *In which position did Michael Schumacher finish the 2004 European Grand Prix?* are types of questions and relations that the initial analysis tries to detect. This relation, though not present in the ontology, or properties related to it would have to be retrieved using a blank node query, and in such a case a special handler is needed.

Depending on what types of resources recognised in the question and what answer types that are expected, the sub-functions are called using the following rules, in the presented order:

1. If there are at least one `FormulaOneRacer` entity and at least one

FormulaOneSeason entity, the handleRacerSeasonNode function is called.

2. If there are at least one FormulaOneRacer entity and at least one GrandPrix entity, the handleRacerGPNode function is called.

3. If there are at least one FormulaOneSeason entity, and the expected AT is either a FormulaOneTeam or FormulaOneRacer, the handleDefault function is first called. Each of the retrieved queries are run to the KB in order to collect an answer. If there are no answers retrieved, then the handleSeasonGP function is called.

4. In other cases, the handleDefault is called.

handleRacerSeasonNode: Handles queries that concerns relations between a FormulaOneRacer and a FormulaOneSeason. Except for predicates that directly links these two classes, relations and related properties are expressed through a blank node, hence the need for a special handler. All the resources from the given input are checked for class types. Instances of the two classes are stored in separate lists. The predicates are added to a list of properties. If a list is empty after processing the input variables, the "empty list" is replaced with a SPARQL variable.

```
for d in drivers:
    for s in seasons:
        for p in props:
            query_prop = "?node %s ?x ." % p

            query_nt = """
                        %s f1p:hasRecords ?node .
                        ?node f1p:season %s .
                        %s rdf:type f1o:FormulaOneRacer .
                        %s rdf:type f1o:FormulaOneSeason .
                        %s
                        """ % (d, s, d, s, query_prop)

            query = """
                        SELECT %s
                        WHERE {
                            %s
                        }
                        """ % (selectVariables, query_nt)
            QUERIES.append(query)
...
return QUERIES
```

Listing 3.2: Excerpt of function handleRacerSeasonNode

Further on, the function constructs multiple SPARQL queries using a nested loop, such that there is one query generated for each combination of drivers, seasons and properties. An extraction of the source code is shown in Listing 3.2. An example of a question that would fall into this handler is the following:

Example: In what position did Lewis Hamilton end the 2014 season?[6]

The result would be similar to the query shown in Listing 3.3. Prefixes are included for demonstration purposes only.

```
PREFIX f1r: <http://f1answers.fake/resource/>
PREFIX f1p: <http://f1answers.fake/property/>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?x
WHERE {
    dbr:Lewis_Hamilton f1p:hasRecords ?node .
    ?node f1p:season f1r:2014_Formula_One_Season .
    ?node f1p:position ?x
}
```

Listing 3.3: Example of SPARQL query

`handleRacerGPNode`: Handles queries that concerns relations between a Formula One Racer and a Grand Prix, that cannot be expressed directly between the two classes. Similar to `handleRacerSeasonNode`, this function will add predicates as properties, check the types of resources and add them to either a list of racers or Grand Prix, before finally creating multiple SPARQL queries, one for each driver, Grand Prix and property. A question that falls into this function is for example *How many points did Nico Rosberg get in the Canadian Grand Prix in 2014?*.

`handleSeasonGP`: Handles queries for `FormulaOneSeason` using the `GrandPrix` class, in those cases the property cannot be directly expressed. This function is also built up similar as the two previous functions described. An example that falls into this handler is the following question: *List all the winners in the 2011 season*. To answer this, the QG component must recognise the winner of each GP that was a part of the 2011 season. The queries created are in the format of:

```
SELECT ?gp ?x
WHERE {
    ?gp f1p:partOf f1r:2011_Formula_One_Season .
    ?gp dbo:firstDriver ?x .
}
```

[6]Some RDF triples have been omitted in these examples to ease the reading

`handleDefault`: Handles all the requests that don't fall into any of the other sub-functions. This function creates multiple SPARQL queries, two for each combination of resources and predicates. One of these will have the resource as the subject of a RDF triple, and the other query puts the resource as the object of the triple. Each query generated by the `handleDefault` function consists only of one RDF triple, and only one select variable. The source code of this function is shown in Listing 3.4.

```
def queryGeneration(PREDICATES, RESOURCES, ANSWER_TYPE_KB):
    ...
    def handleDefault():
        QUERIES = []
        selectVariable = "?x"

        # For now, can only create queries if a list of resources
            and a list of predicates are given
        # To be improved in the future
        if not RESOURCES:
            return []
        if not PREDICATES:
            return []

        # Creates two queries; One query has the resource as the
            subject, the other query has the resource as the
            object
        for r in RESOURCES:
            for p in PREDICATES:
                qt = "<%s> <%s> %s ." % (r, p, selectVariable)
                query = "SELECT %s WHERE {%s}" % (selectVariable,
                    qt)
                QUERIES.append(query)

                qt = "%s <%s> <%s> ." % (selectVariable, p, r)
                query = "SELECT %s WHERE {%s}" % (selectVariable,
                    qt)
                QUERIES.append(query)

        return QUERIES
```

Listing 3.4: Excerpt of function handleDefault

3.4 Full example

There are many different questions that could be demonstrated in order to show strengths and weaknesses of the different components, where a question fails and

how one component affects another. However, the purpose of this run-through example is to show a full demonstration of the QA components, including QG and answer retrieval. Hence, it's useful to use an example that can be correctly processed through the pipeline and answered. Further analysis of the output and results of each component will not be discussed in this section, but a full evaluation is reviewed in Chapter 4.

The question used for this demonstration is one of the questions used for evaluating the system, and was also mentioned as an example under QG, Section 3.3.5.

CQ19. List all the winners in the 2011 season.

Note that this is a type of question that is first handled by the default handler in QG, but are sent to a special-case function afterwards. See case 3. in Section 3.3.5.

3.4.1 Phrasing

Using the NLTK word tokenizer, the question is split into the following tokens:

['List', 'all', 'the', 'winners', 'in', 'the', '2011', 'season']

These are further tagged using the NLTK POS tagger. The words are listed below along with the POS tag, in addition to a short description of the tags collected from the Penn Treebank POS Tag list [26].

List, NN - Noun, singular

all, PDT - Predeterminer

the, DT - Determiner

winners, NNS - Noun, plural

in, IN - Preposition

the, DT - Determiner

2011, CD - Cardinal Number

season, NN, Noun, singular

Figure 3.5 shows how the words are chunked into phrases, using the list of predefined patterns described in Section 3.3.1.

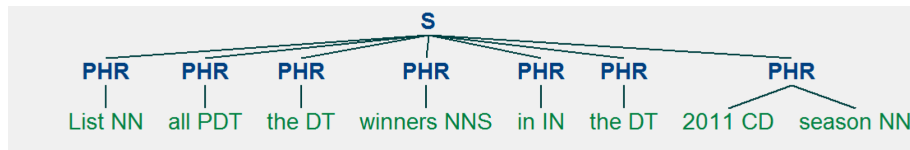

Figure 3.5: Result of words chunked to phrases

As shown in the figure, only the words "2011" and "season" were chunked into a phrase consisting of more than one word. The other phrases consisted only of single words. "2011 season" was matched to the regular expression (<CD>|<JJ.?>)*<NN.*>*.

3.4.2 Answer Type Detection

After splitting the question into phrases, these phrases are sent to the ATD component. The function loops through the words, and detects the QP "List". Because this QP needs to be processed further, the ATD component loops through the remaining words and looks for phrases that contains nouns. Utilising the POS tags assigned to the words, the component detects "winners" as the first phrase that potentially can be the expected AT. This phrase is then looked up in the lexicon of phrases, which in turn returns two possible results. Both `firstDriver` and `FormulaOneRacer` are mapped as potentially semantic items. However, only one of them is a class (`FormulaOneRacer`). Because the component is looking for ontology classes, it selects `FormulaOneRacer` as the final interpretation of AT.

3.4.3 Semantic Phrase Detection

For each phrase constructed by the Phrasing compoment, this step checks if the phrase is equal to one of the entries in the lexicon of phrases. Only the term "winners" was detected as a potentially semantic phrase.

3.4.4 Semantic Phrase Mapping

The results for mapping the term "winners" are the same as described for ATD, `firstDriver` and `FormulaOneRacer`. However, note that in this case it's the predicate `firstDriver` that's interesting to capture.

3.4.5 Named Entity Recognition

The pipeline continues by finding possible NE. The NER component discards phrases that have been mapped to a semantic item in the previous step, i.e. "winners". In addition, the NER component checks if there exists at least one noun in the phrase, in order to include the phrase as a NE candidate. The results of NER is that only the phrase "2011 season" is detected as a candidate.

3.4.6 Named Entity Disambiguation

NED checks the lexicon of resources, and compare the phrase "2011 season" to all the entries in the lexicon, before returning the URI of the entity that yielded the highest similarly score. The returned entity from this step was `http://f1answers.fake/resource/2011_Formula_One_Season`, which indeed is a correct mapping. The similarity score was 0.69.

3.4.7 Query Generation

As already reviewed in Section 3.3.5, the QG component checks the type of resources detected in the question. Because a specific `FormulaOneSeason` is detected, and the answer type is expected to belong to the `FormulaOneRacer` class, the `handleDefault` handler is first called. However, there were no answers retrieved from the queries generated in this default handler. This is because there was no direct relation between a `FormulaOneSeason` and a `FormulaOneRacer` through the predicate `firstDriver`. In this case, again as described earlier, the protocol is to use the `handleSeasonGP` function.

Because the input data only consisted of one property, one season and zero GP instances (replaced by one variable), the number of different SPARQL queries can be calculated as $1 * 1 * 1 = 1$. The full query is shown below.

```
PREFIX f1r: <http://f1answers.fake/resource/>
PREFIX f1o: <http://f1answers.fake/ontology/>
PREFIX f1p: <http://f1answers.fake/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>

SELECT ?gp ?x
WHERE {
    f1r:2011_Formula_One_Season rdf:type f1o:FormulaOneSeason .
    ?gp f1p:partOf f1r:2011_Formula_One_Season .
    ?gp dbo:firstDriver ?x .
}
```

Listing 3.5: Final query of full example

After running the query towards the KB, a list containing five URIs is retrieved. All of these entities are then checked against the expected AT. To ease the reading, only the labels are shown below.

Sebastian Vettel, Lewis Hamilton, Fernando Alonso, Jenson Button and Mark Webber.

All the entities were checked and mapped to a `FormulaOneRacer`, hence all the entities are returned as the "expected" correct answers.

3.5 Prototype of Web Interface

The following section presents a first version prototype of the QA system, using a simple web interface. The interface consists of one web page with a search box, where a user can enter the question. Simple fault handling like checking for "empty input" has also been implemented. The framework is constructed using Flask[7] version 0.12, a micro-framework for Python. With Flask, both the backend and the frontend (web interface) of the program are constructed through a combination of python code and common HTML syntax, where it's possible to include python code inside the HTML code. A snapshot of the start page is shown in Figure 3.6 along with a sample question.

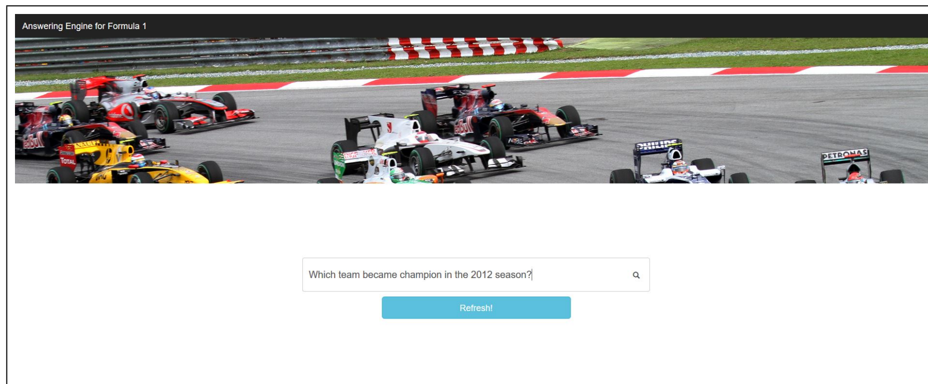

Figure 3.6: Web interface with a sample question

The user also has the option of getting a random question by the click of a button. These sample questions have been checked manually, to ensure that they can be answered by the system. Of course, the user is free to modify the question. One of the example question is "Who won the 2011 Australian Grand Prix?", but this can modified to e.g. "Who won the 2006 German Grand Prix?". The purpose of the sample questions is to provide some ideas and tips to the users on which kind of questions to ask and what kind of knowledge the system can provide. But they are still free to come up with own questions.

The answers are shown right below the search box, providing a direct response to the user. When the answer is a resource, the answering module takes care of displaying the literal value (label). The labels link to the knowledge base page of the returned resource(s). The results of the underlying process are shown under a hidden tab, but can be displayed by expanding the tab (+ icon). Figure 3.7 demonstrates this.

[7] http://flask.pocoo.org/

Answers

Red Bull Racing

Processing Information +

Chunked Sentence

Which team - became - champion - in - the - 2012 season - ?

Answer Types Detected

team

Answer Type Classes

http://dbpedia.org/ontology/FormulaOneTeam

Named Entities

Label	URI	Score
2012 season	http://f1answers.fake/resource/2012_Formula_One_Season	0.690841554013

Mapped Phrases

Phrase	Semantic Items
champion	http://f1answers.fake/property/constructorsChampion, http://f1answers.fake/property/driversChampion

SPARQL Queries

SELECT ?x WHERE {<http://f1answers.fake/resource/2012_Formula_One_Season> <http://f1answers.fake/property/constructorsChampion> ?x .}

Figure 3.7: Presenting the results

Chapter 4

Evaluation

Evaluating the system is necessary in order to analyse and present a qualitative measure of the system's performance and results, but also to discover weaknesses and potential improvement points. The evaluation carried out and presented in this chapter includes:

- Evaluating the system end-to-end
- Evaluating the components separately
- Evaluating the ontology and the knowledge base
- Analysing types of errors

We start by first describing the background and methodology of the evaluation process, before the next section presents the results.

4.1 Methodology

4.1.1 Baseline questions

The evaluation is based on a set of 50 questions, referred to as "the baseline questions". The construction process of these is discussed in Section 5.2. Unlike some other systems, e.g. as DEANNA by Yahya et al. presented in this paper, the evaluation is not based on a public dataset of general questions like the QALD series[1]. The baseline questions were constructed manually, specifically targeting the domain knowledge of Formula One. The overall goal was to generate a set of questions that first of all were interesting and relevant, but also included a variety of different questions that would challenge the QA system.

The questions were written in "natural language" in English. This means that we have tried to avoid uncommon terms and phrases that are unlikely to be expressed in a question by the common user. This is again one of the main

[1] `https://qald.sebastianwalter.org/`

challenges and goals of the QA system; understanding and interpreting a NL question. For example, the domain-specific vocabulary uses the term *first driver* to address the winner of a Grand Prix, but it is more natural to use the term *winner*. However, we have assumed that users will have some knowledge about Formula 1 and terms that are related to the sport itself. For instance, they should know how to express and refer to a specific Grand Prix by including the location and year. Phrases such as "pole driver" or "pole sitter" should also be familiar. Additionally, one of the guidelines we used when constructing the questions was that they should be written in the same format as one would ask the question oral. This is also one of the factors that separates a question from a "keyword query".

In order to represent a qualitative result, it's important to have a variety of questions that seek different kinds of answers. When we constructed the baseline questions, we exploited the ontology and looked for different kinds of properties and relations that could be translated into a question. This includes distinct questions that are completely unrelated, but also similar questions that were formulated differently. For example, the question *Who won the Australian Grand Prix in 2012?* is similar to the question *Who was the winner of the 2012 Grand Prix in Australia?*, but it's interesting to include both of these questions in the evaluation because it challenges the system to interpret different formulations. However, we have tried to minimise the number of similar questions and put more effort into having different types of questions. We have included a majority of questions that can be answered by the ontology/knowledge base, in addition to a few that can't be answered. This is again to see how the system handles questions outside of it's domain.

The complete list of the 50 baseline questions is added to Appendix A.

Category	Description	Question count
Factoid	Only one answer	35/50
List	List of answers	15/50
Type 1	Easy, consists of max. 2 RDF triples	22/50
Type 2	Simple, more than 2 RDF triples. Can include filtering, ordering and limitation	14/50
Type 3	Aggregated questions with counting, summation, averaging and grouping	10/50
Type 4	Data not available or ontology is insufficient. Can't be answered.	4/50

Table 4.1: Categories and types of baseline questions

Chapter 3 stated that the implementation of this QA system will only con-

sider factoid and list questions. Hence, the baseline questions have been categorised into either **factoid** or **list**, and consist only of these types. This makes it possible to evaluate the categories separately. For each question in the dataset, we manually created a SPARQL query that could retrieve the answer from the knowledge base. The purpose of this is first to check that the knowledge base is able to answer the question. Secondly, we need to collect the *ground truth* answers in order to compare them to the answers retrieved from the system. Finally, we introduced a second categorisation (types) of the baseline questions that is based on the difficulty of the SPARQL query. By categorising the questions into types of difficulty, it's possible to evaluate the end-to-end system more precisely. This will be further discussed after the results presentation. For an overview over the categories, types and the criteria of SPARQL difficulties, see Table 4.1. Note that there are two orthogonal dimensions used here. Each question is assigned both a type (1-4) and a category (factoid or list), not just one of them.

Figure 4.1 below presents an extraction of the 50 baseline questions, including the category, type, SPARQL query and the ground truth answers.

Question	Type	Category	SPARQL	Answers

Who won the Australian Grand Prix in 2011?	1	FACTOID	SELECT ?x
			WHERE {
			dbr:2011_Australian_Grand_Prix
			dbo:firstDriver ?x .
			}
				http://dbpedia.org/resource/Sebastian_Vettel

Which drivers had the pole positions in the 2011 Season	2	LIST	SELECT ?x
			WHERE {
			?x f1p:hasRecords ?node .
			?node f1p:season
			f1r:2011_Formula_One_Season .
			?node dbo:poles ?pole .
			FILTER(?pole > 0)
			}
				http://dbpedia.org/resource/Lewis_Hamilton
				http://dbpedia.org/resource/Mark_Webber
				http://dbpedia.org/resource/Sebastian_Vettel

What is the average points Lewis Hamilton gained in all seasons?	3	FACTOID	SELECT (AVG(?p) as ?avg)
			WHERE {
			dbr:Lewis_Hamilton
			f1p:hasRecords ?node.
			?node f1p:points ?p.
			FILTER(?p>0)
			}
				224.7

Who is the youngest driver to become a champion?	4	FACTOID	N/A	N/A

Figure 4.1: Extraction of baseline questions

4.1.2 Ontology and KB Evaluation

Since the focus of this paper is related to the implementation of Question Processing and not the underlying data or ontology, only a brief insight of ontology and KB evaluation will be presented. The paper by Pirbazari presents a more detailed analysis of these parts, and readers are referred to that paper for the results.

As described in the previous section, we created a SPARQL query for each of the baseline questions. If it was possible to create a query, run it and retrieve an answer, it means the ontology is sufficient to answer the question. However,

we have deliberately omitted too many questions that are not covered by the ontology. The reason for this is because the purpose is to see how well a natural language question can be parsed into a query, which can be used to collect the, hopefully correct, results. As opposed to this, evaluating questions that are not covered in the domain is not of high interest. We have however included a few just to check the response of the system.

Because we know beforehand which questions that can be answered by the ontology, evaluating the ontology based on how many of the baseline questions it can answer is unnecessary. However, the results might give an indication to which relations that are difficult to interpret and match, or which relations or properties that should be added or removed in order to make it easier to parse the question into a query.

For example, if a SPARQL query is complex because there is no simpler relation between e.g. two classes, the results might indicate that there is a need to implement this relation.

To sum up, evaluating the ontology does not simply consist of checking which of the baseline questions that can be answered, because this is already determined when we created the questions. It also includes discussing weaknesses and improvements in the ontology that would make it easier to convert a question into a query. This may consist of improving relations that are already present, or adding relations and/or properties that are missing, according to relevant questions.

For the KB, we present some statistical information about the data.

4.1.3 End-to-end Performance

The end-to-end evaluation consists of running each baseline question through the QA program, collect the answers and compare them to the ground truth. This means that we do not consider the performance of the individual components in the QA system, but only the returned answers. Based on the comparison to the ground truth, the questions are categorised into 1 of 4 Question Evaluation Categories (QEC), described in Table 4.2 below.

Category	Description
QEC-1	Correctly answered
QEC-2	Incorrectly answered
QEC-3	Partially correct answered
QEC-4	No answers returned

Table 4.2: Table of Question Evaluation Categories (QEC)

When evaluating the results and comparing the returned answers, we con-

sider the URIs for resources, and not for example labels. Distance metrics are compared on the numeric value, and not along with units. This is because the evaluation doesn't concern answer preparation and presentation. Unless specified in the question, if there are multiple answers returned it doesn't matter in which order they are presented.

QEC-1 means that there is 100% match between the expected (relevant) answers and the retrieved ones. None of the answers returned are irrelevant, and all the relevant answers were retrieved.

QEC-2 means that the question has been answered, but the returned results are incorrect. There are no answers returned that is relevant or correct.

QEC-3 is the category for questions that are answered partly correct. This means that there is at least one answer that is correct, and at least one answer that is incorrect or missing. There is no further analysis of how many of the returned answers which are correct or incorrect, just that there is at least one of each. This category will only concern List questions.

QEC-4 is the category for questions that weren't answered at all, meaning that there were no answers returned. Note that questions of type 4 (questions that can't be answered) will also fall into this category if there are no returned results.

4.1.4 Analysing the components

Component evaluation checks each component in the QA system individually, regardless of the returned answer. For some of the components, as described below, the evaluation is based on a subjective judgement and not "hard evidence". In this evaluation process, we mainly use a binary method where a component either fails or succeeds. However, for some components it's also useful to consider what is known as the confusion matrix, where we distinguish between true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). These terms are defined differently for different components, and will hence be presented in the results section for those components where it's relevant.

Phrasing

Because there are no further processing after tagging a question with POS before chunking the question into phrases, they are considered as one step (called phrasing). Evaluating this component is based on a subjective judgement, however with some guidelines. If the sentence is chunked into semantic phrases such that relevant predicates, resources and other phrases can be retrieved and mapped, then we consider this component as successful. If the chunking process leads to phrases that contains more than one semantic item, or prevents a semantic item to be parsed, it is considered as a failure.

Answer Type Detection

As described in Chapter 3, determining the answer type is 2-step process. First, based on the question phrase, the system tries to determine a list of candidates out of a predefined list of answer types. For example, this answer type could be a number, a location or a person. If that type can further be mapped to a class from the ontology, e.g. a racer or a team, then this is carried out and the expected answer type is that class. Otherwise, we denote the expected answer type as a number, date, string etc., which are not classes from the ontology.

For each of the baseline questions, the expected answer type determined by the program is compared to the ground truth (manually determined). If the answer type is a non-class and correctly detected, we denote the component as successful. If the answer type is a class from the ontology and the program correctly identifies this class as the answer type, we denote the component as successful. Otherwise, the component has failed.

NER

NER is more or less determined by the Phrasing component, however, there is one further step used in this component. If a phrase contains at least one noun, it may be considered as an entity. Otherwise, if there are no nouns in the phrase, it is not considered as a candidate for a NE.

Evaluating NER is carried out with the following rules: If a NE is not detected (false negative) or if the system recognises a NE where there are none (false positives), we denote the step as failed. Otherwise, the step succeeded.

NED

NED evaluation considers the mapping of a NE phrase to the NE's URI. If the component returns the correct URI for the NE, then the step succeeded. If it fails to map it or returns an incorrect URI, then this step failed. Because the NER component may detect phrases that are not a NE, this could lead the NED component to try and map phrases to URIs, even though it shouldn't. We have also denoted these cases as a failure. This means that the NED component's evaluation score is both affected by itself, as well as the result of the NER component.

Phrase Detection

Again, phrases are determined by the first component, Phrasing. The evaluation of phrase detection consists of recognising phrases that should/could be mapped to a semantic phrase (i.e. mapped to a semantic item in the phrases-lexicon). If all relevant phrases are detected, then the step is successful. If at least one phrase is not detected, then we denote the step as failed, because every phrase can be important for the query generation.

Phrase mapping

Phrase mapping evaluation is based on the accuracy of mapping phrases detected in the previous step to the semantic phrases (i.e. predicates or classes) in the lexicon. This step can fail if the phrase cannot be mapped to the relevant semantic phrase, or as a result of incorrectly detected phrases from the previous step. Because a phrase can potentially be mapped to a number of different semantic items, it is not incorrect to return multiple semantic phrases. This is however a judgement call, and will be further discussed in Chapter 5.

Query generation

Along with the baseline questions, we added only one SPARQL query for each question. However, there is not necessarily just one correct query that can retrieve the correct result, and for every correct query there isn't one that is more correct than another. Hence, there is no point in comparing a query generated from the system to one created beforehand.

Evaluating the component that generates queries is more difficult than the other components. This is because it is the last step in the pipeline, and may be affected by any of the other components. We decided to evaluate this component on the same principles as the other components; even though it fails because the failure of a previous component, we denote a failure in this component. However, it is also possible that none of the other components failed, but the query generation component failed alone.

By failing in this step, we mean that it failed to generate a query that captures the correct predicates, resources and possibly the returned answers. In the previous components, it was possible to perform the evaluation without considering the returned answers. However, in this case it's difficult not to consider them.

There are two other ways we could have evaluated this step. First, we could have denoted a failure only if this component alone was responsible for a failure. Second, we could have determined that a correct query is a query that returns any result, and a failure means that there were no queries generated. Generated queries that don't return any results are removed during the query generation process, and is therefor not considered as a part of the result of this component. Our decision is again a judgement call, discussed further in Chapter 5.

4.2 Results

4.2.1 Knowledge base statistics

The statistical information about the KB presented here provides an insight of what type of knowledge it covers and how much data that is present.

The KB consists of approximately 81 000 RDF triples, divided into two main schemes; triples related to the ontology design and triples related to instances.

There are a total of 2152 instances defined in the KB, distributed on six classes. This distribution is shown in Figure 4.2.

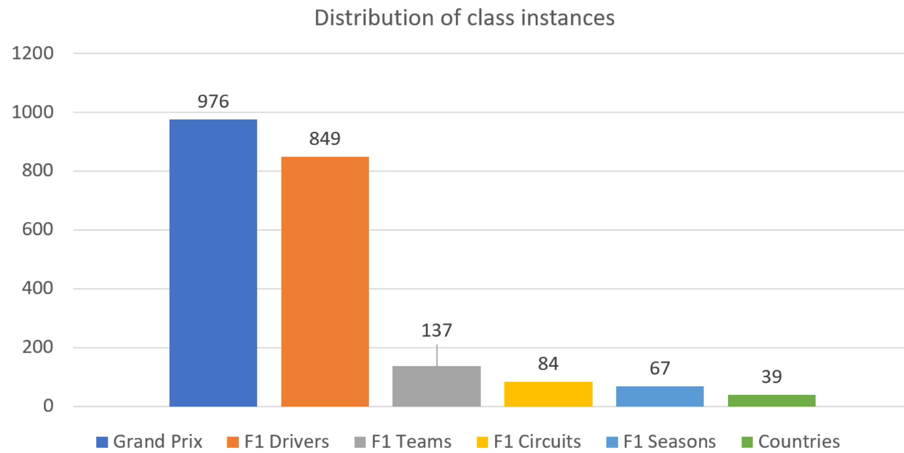

Figure 4.2: Distribution of class instances

The results shows that there is a clear asymmetric distribution. About 84% of the resources belongs to the classes Grand Prix or Formula 1 drivers, but this shouldn't come as a surprise.

First of all, the KB covers a time span of 67 years/seasons, which is a relative long time. Within each season, there is a certain number of Grand Prix. In the last 30 years, this number has been in the range of 15-20. On average there has been about 14 Grand Prix hosted each season since 1950. By summing up these numbers, we end up with a great amount of Grand Prix instances relative to the other classes.

The same logic applies for drivers. Given a period of almost 70 years, it's only natural that there are a lot of different competitors listed. Athletes that compete on the highest level in their sport are expected to deliver good results in order for them to continue on the highest level.

One could expect that the number of different teams would also be in the same range as Grand Prix- and driver instances. However, teams do not have an "expiration date" as drivers have. Just like with the history of football, basketball, baseball, hockey and other world-known sports, teams survives through decades and centuries. For example, both Ferrari and Mercedes has competed in Formula One since the 1950's [7].

Figure 4.3 below shows how the number of competing drivers and the number of races per season, has evolved from 1950 to 2016.

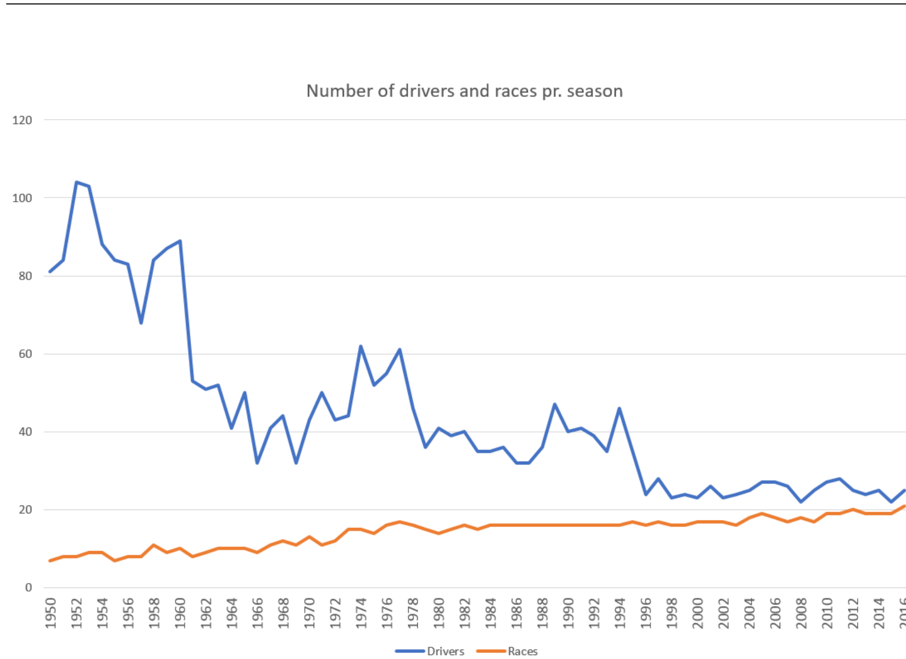

Figure 4.3: Drivers and races per season

4.2.2 Ontology

The baseline questions included 4 questions that could not be answered. In this subsection we discuss why these can't be answered. In addition, we discuss some topics that is related to ontology design, which affected the performance of the system. We refer to questions by their competency question (CQ) number.

- CQ40. Which driver had the most consecutive pole positions since 2010?
- CQ41. Who is the youngest driver to become a champion?
- CQ42. Who is the youngest driver with more than one win?
- CQ43. How many points did Nico Rosberg get in the Canadian Grand Prix in 2014?

Questions CQ41 and CQ42 both look for the *youngest driver*, which means that we must be able to sort drivers by their age/birth date. The property "birth date" has been included in the ontology design, but the data has not been entered. Hence, these questions cannot be answered due to missing data.

Question CQ43 asks for a property for a given driver for a given Grand Prix. This information is not available, neither as a relation in the ontology or as data in the KB.

Question CQ40 on the other hand asks for data that is available in the KB, but it is not possible to retrieve this information through one SPARQL query.

This question falls into the category of "aggregation problem". In order to answer this problem, we would have to loop through all the Grand Prix since 2010, check which driver who was the pole driver, keep a score of the longest streak of consecutive pole positions, before returning the driver with the longest streak. In addition to checking the longest streak within each season, we would also have to consider streaks that crosses seasons. For example, if a driver was the pole sitter of the last race of season 2011, in addition to the first race of season 2012, then this should also be included in the streak.

These types of questions have not been considered in this implementation, and hence cannot be answered.

Below we discuss two other questions that failed to be answered because of ontology weaknesses, even though the data was available in the KB.

- CQ11. List the teams that participated in the 2015 season.
- CQ25. Name a circuit in Spain.

Question CQ11 asks for a relation that is not easily expressed in the ontology. An improvement point here could be to include a direct relation between a team and a season, for example through a predicate called "participateIn".

Question CQ25 asks for the location of a Formula 1 circuit, but this relation is not directly expressed in the ontology. In order to find the answer(s), we would have to modify the query generation component to look for relationships between countries and circuits, through the Grand Prix class. An improvement point for the ontology here would be to include a direct relation between circuits and the location of them.

4.2.3 End-to-end Evaluation

The results for the end-to-end evaluation of the system is presented below. Table 4.3 presents the evaluation categories (see Table 4.2) and how many of the 50 baseline questions that fell into these categories.

Category	Factoid	List	Type 1	Type 2	Type 3	Type 4	Total
QEC-1	18	4	18	4	0	0	22
QEC-2	3	2	1	0	3	1	5
QEC-3	0	0	0	0	0	0	0
QEC-4	14	9	3	10	7	3	23
Total	35	15	22	14	10	4	50

Table 4.3: Results table of end-to-end evaluation

As shown in the table, there were none questions that fell into the QEC-3 category, which means no questions were partly answered correct. Figure 4.4 omits this category and presents the results for each of the four question types, distributed on QEC, with respect to percentage numbers.

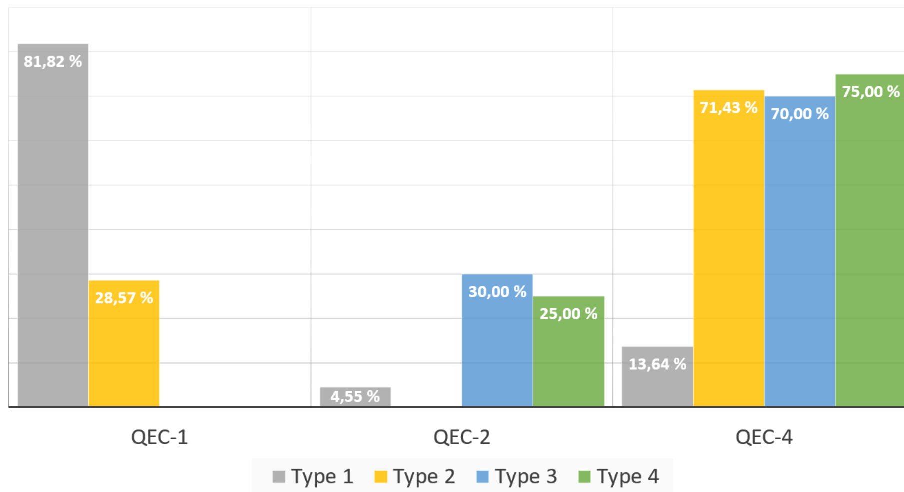

Figure 4.4: End-to-end evaluation for question types, distributed on QEC

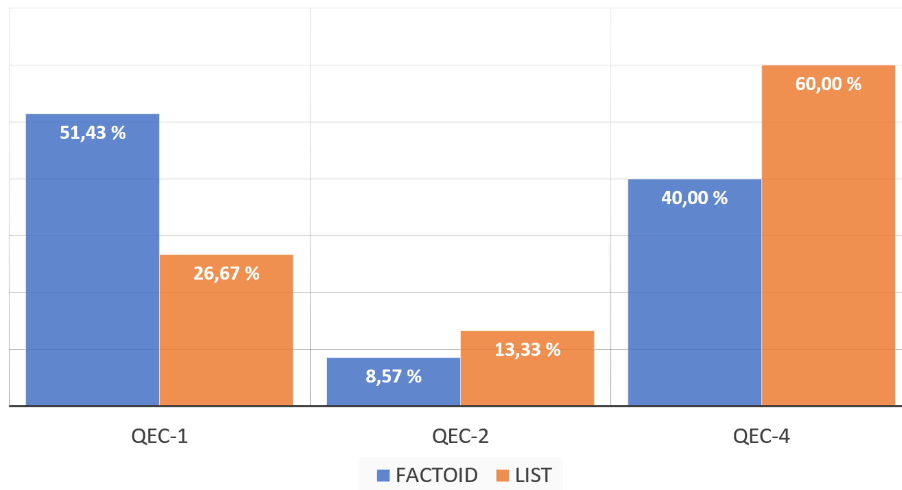

Figure 4.5: End-to-end evaluation for factoid and list questions, distributed on QEC

Figure 4.5 presents the same results for the categories factoid and list. The results indicates that we answer correctly on approximately 50% and 25% of factoid and list questions, respectively. For type 1 questions, which is considered the easiest questions, we answer about 8 out of 10 questions correct. Type 2, 3 and 4 questions were not answered about 70-75 % of the time. The difference between these is that we answer correctly on the remaining questions of type 2, where as the remaining questions for type 3 and 4 were answered incorrectly.

Overall, the QA system answered correctly on 22 out of 50 questions, which yields an accuracy score of 44%. Table 4.4 summarises the overall accuracy for the different question categories and types.

Total	Factoid	List	Type 1	Type 2	Type 3	Type 4
0.44	0.51	0.27	0.81	0.29	0	0

Table 4.4: Overall Accuracy

4.2.4 QA Components Evaluation

The following section presents the results of the performance for each of the components in the QA system pipeline.

Figure 4.6 shows the accuracy for each component for each of the 4 question types. The figure shows that the components have a high accuracy for type 1 questions, which is also reflected by the end-to-end evaluation in the previous section. Further on, the results indicates that the accuracy of the components decreases as the question difficulty increases.

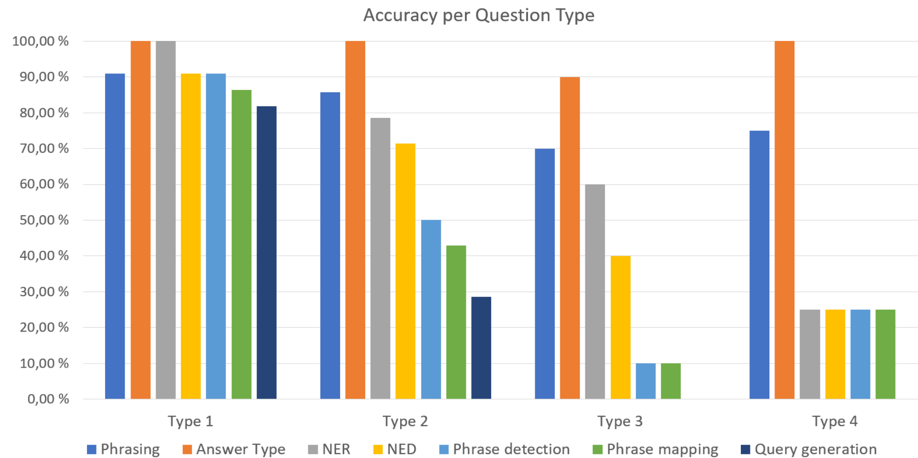

Figure 4.6: Accuracy per Question Type

Figure 4.7 presents the overall performance for each component. The results indicate a clear, decreasing success-rate when moving "down the pipeline". As discussed, we chose to define a failure in one component, even though it may have failed because it was relying on another component that failed. This "pipeline effect" will be further discussed in Chapter 5.

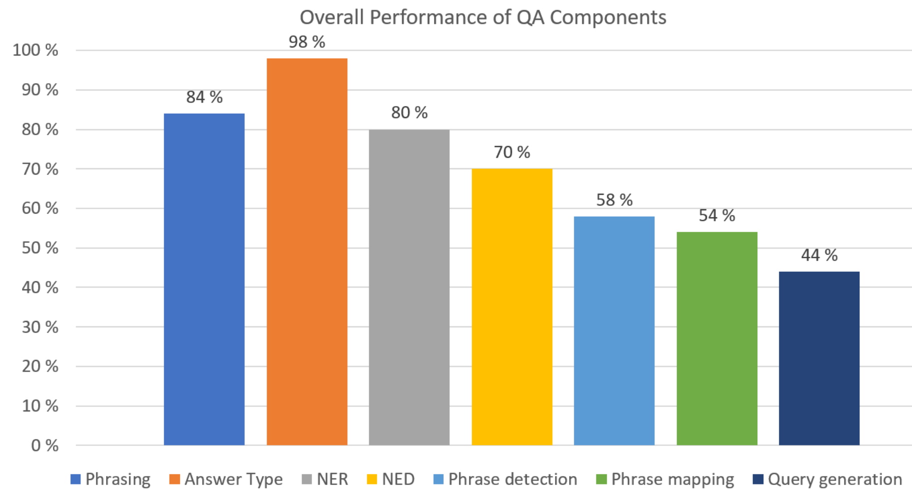

Figure 4.7: Overall Performance of QA components

Most components were evaluated using a binary metric; either it succeeded or it failed. However, for NER and NED it was possible to use the confusion matrix [6], and calculate the precision, recall and F1-scores. The results are summarised in Table 4.5 below.

	NER	NED
True positives	40	40
False positives	16	16
False negatives	0	0
True negatives	NA	NA
Precision	0.714	0.714
Recall	1.0	1.0
F1-score	0.833	0.833

Table 4.5: Precision, recall and F1-score of NER and NED

In total there are 40 NEs mentioned in the 50 baseline questions. All of these were correctly recognised and mapped, yielding a recall score of 1.0. However, there were also 16 false positives, phrases that were incorrectly recognised and mapped as NEs. This reduced the precision to 0.714.

In addition, 5 concepts are mentioned in the set of questions. These were recognised but not mapped due to missing references in the ontology/KB, and therefor we do not consider concepts in these results. True negatives are not considered because every phrase that is not meant to be an entity would then have to be considered, which is superfluous.

All in all, the answer type module is the best performing component with an accuracy of 98%.

4.2.5 Error Analysis

Further analysis of the evaluation results with regards to *why* some components failed, revealed that there were some types of errors or faults that occurred frequently. With this analysis we try to determine if there are specific fields for some of the components that could be improved, such that the overall performance and accuracy will increase. We divided these *error types* into 11 categories, among them the components of the QA system. Table 4.6 summarises the results, with the type of errors and how many questions that are related to them. With these results we are interested in finding the root cause for errors, so components which failed because of other failures are not specifically included. However, for some questions it was possible to determine that they had major faults in more than one component, hence they have been included in the count of more than one category.

Errors related to ontology design weaknesses and lack of data in the KB has already been reviewed in the beginning of the results section. Among the other categories, there are two specific categories that are worth reviewing.

20 questions included statistical terms, ordering or limitation attributes like "most", "least", "since", "shortest". These terms have not yet been addressed in the program, and the QA system cannot distinguish between "most points" and "points", for instance. Without doubt, this error type stand out among the others by far. Solving these faults would probably increase the overall accuracy more than any of the other error types. Some examples related to these categories are

CQ45. Which race was the shortest?

CQ46. Which season had the most races?

Another interesting error type is concept mapping. A concept is a collection of entities that are grouped together on some attributes, but is not an entity itself. For example, the concept of "Italian Grand Prix" should include a list of all the Grand Prix hosted in Italy. The concept of "German drivers" should include a list of all Formula 1 drivers from Germany. Concepts have not been addressed

Error type	Count
Statistical terms	20
Concept Mapping	4
Ontology Design	4
Semantic Phrase Mapping	4
Semantic Phrase Detection	3
KB data missing	2
Phrasing	2
Query Generation	2
Answer Type Detection	1
Named Entity Disambiguation	1
Named Entity Recognition	1

Table 4.6: Types of errors and the related frequency of questions

in this implementation, and hence the program fails to recognise these. Even though there are not many questions related to this error type, it's natural to include concepts in questions. If the system was tested on a larger dataset, it's not unlikely that more questions would fall into this category. Some examples from the baseline questions are:

CQ30. List all the german drivers in the 2015 season

CQ37. List all the teams who have won the Italian Grand Prix more than once

Chapter 5

Analysis and Discussion

The answering engine presented in this paper is a first-version implementation, and there is no doubt that it will have flaws and improvement points. The following sections of this chapter present and discuss challenges, weaknesses and strengths of the program associated with the Question Processing module and the evaluation results. The contribution of this paper does not include improvements of the ontology design or the knowledge base.

5.1 Precautions

A full-developed system should not inflict many restrictions on the user, but in this system we take some precautions.

First of all, it's expected that users are familiar with the concept of Formula 1, common terms within this sport and how to refer to Named Entities (NE). This means that teams and drivers are mentioned by full name (e.g. "Lewis Hamilton" in stead of "Hamilton"), and specific Grand Prix are mentioned with both location, year and the term "Grand Prix". For example, the question "Who won in Italy in 2014?" does not specifically refer to the 2014 *Grand Prix* in Italy, and questions in this format are unlikely to be answered correctly. However, questions that refers to concepts like "Italian Grand Prix" are allowed.

It's not expected that users are familiar with the domain-dependent ontology language, such as the names of the predicates. After all, the answering engine should support natural language questions.

We assume that questions are constructed using one of the common question phrases, such as "who", "what", "where", etc. It's not specifically necessary to have the wh-word as the first word in the question, for example the question phrases "In what country ..." and "At which position" will be handled by the program. However, the program is not designed to interpret questions phrases of the type "Can you tell me...", "Please give me a list of..." and "I want to know...", even though it potentially may handle the question correctly in some cases.

5.2 Baseline Questions

When we constructed the set of baseline questions, the idea was to create different types of questions that looked for different types of answers, addressed different predicates from the ontology and in general asked for different things. Naturally, there are different ways on how to ask a question, but we tried to formulate them using a natural language that made sense for us. We also included a few questions that asked for the same relation (e.g. winner of a race) but were formulated differently. We believe all the questions are relevant for the domain knowledge (except the ones that specifically are not meant to be answered) and are appropriate to use as a baseline for evaluation. There are many properties and relations from the ontology that are not specifically addressed through the baseline questions, and with more time it would be ideal to have a much larger set of questions that included them as well.

We categorised the questions into either factoid- or list questions, which was the intended domain of question types to include. However, we needed to collect the results before we could categorise them. The reason is that some questions are formulated as a standard factoid question, but returns multiple answers (and vice versa). For example, questions including statistical terms like "which team has most wins since 2010?" is intuitively thought to be a factoid question. However, there could be multiple teams sharing the "first place", and in that case it would be considered as a list question. "Name a circuit in Spain" could be a list question if it returned multiple answers, but in our domain it was only one circuit returned, and it was hence categorised as a factoid question.

The baseline questions were also used as training data during the development of the program. It was necessary to have a set of questions that included phrases before implementing those into lexicons. This of course has an impact of the evaluation results, because missing phrases would continuously be added to the lexicons if they weren't present. The reason why we continued using the exact same set of questions for evaluation is because those phrases should have been added to the lexicons either way. In the large picture, we assume that we have a close-to-complete lexicon that includes common phrases and the corresponding semantic items. To simulate this vision, we made sure that semantic phrases that appeared in the baseline questions were present in the lexicon. However, it's uncertain how much the evaluation score would differ if we used another set of questions for testing the system, and it's always a good practice to separate training data and testing data. This is something we will take wisdom of for future research.

5.3 Lexicons

The lexicons of phrases and resources are a major contribution to the program, and is partly responsible for the accuracy of many of the components. Without them, answer types cannot be mapped, semantic phrases and named entities cannot be mapped and in general the whole program breaks down without

them. Hence, it's really important to have a good set of lexicons! However, maintaining these sets is also a big task.

Both lexicons are constructed using a program written in Python, which collects the resources and the predicates from the knowledge base and stores these in separate files. The resource lexicon are more or less complete at this point, and there is no further processing of this file.

The lexicon of phrases on the other hand needs modifications. Specifically, natural language phrases needs to be added to the lexicon for each semantic item/phrase. As of now, this is a manual procedure where common terms and phrases found in the ontology and in the set of baseline questions are added to the lexicon. Clearly, this is not a good approach considering future expansion of the domain.

Future work should include implementing an approach that can collect predicates from the ontology and automatically (or at least semi-automatically) provide a synset of words, for example using the lexical database called WordNet. Synsets in this context refers to a group of synonyms [40]. However, there still might be a need to include a set of domain-dependent terms as well, because libraries such as WordNet often focus on a general, domain-independent vocabulary.

5.4 Components

5.4.1 Phrasing

The phrasing component is one of the easiest components regarding implementation, because it's only constructed using two third party libraries, in addition to a handful of regular expressions. However, it's also one of the most important components because it affects all the other components, and many of the baseline questions failed at least because of a bad chunk. Because the component is constructed quite simple, it also indicates that there is room for improvements. One solution here for future work could be to include a higher quantity of regular expressions as well as more detailed expressions, targeting specific patterns for entities and phrases. Another approach could be to use the lexicon of resources and phrases, and come up with different sets of candidate phrases in stead of only one possible solution.

However, note that the component performs exceptionally good (84%) relative to the implementation simplicity.

5.4.2 Answer Type Detection (ATD)

Questions that seek existential facts, often beginning with "is", "are", "was" or "were" are detected by the ATD module but not processed any further. These kinds of questions were not considered in this implementation. The same applies for explanatory questions beginning with "why". They are included in the ATD module, but only to give a specific response to the user.

Based on the evaluation results, the ATD module is the best-performing component in the system, and there is probably a few reasons why. First of all, assuming that questions begin with a common question phrase, it's not a difficult task to retrieve the consecutive nouns by looping through the words. The second task of mapping those words may be more difficult, but in our case the baseline questions were also used as training data when the program was developed. This made it possible to specifically include all the answer type words in the lexicon of phrases, assuring that they would be correctly mapped to the ontology class. For answer types that were mapped to e.g. a number, it was also easy to update the local look-up table. However, there are two specific challenges associated with this module when looking ahead.

The first part of the module that detects natural language words (that are later mapped to a type) can't handle non-consecutive nouns. For example, the phrase "List races and winners" will only yield an answer type of "races" (e.g. Grand Prix), unless "races and winners" are chunked as a phrase in the phrasing-component. It's debatable whether it's the phrasing-component or the ATD component that is responsible of handling this issue, but it's definitely a task for future work to resolve it.

As explained, the baseline questions were used as training data to tweak and improve the program during development. This made it easy to continuously update lexicons with common terms and phrases. Given a new set of questions, or a larger set of different questions, there would probably be multiple phrases that can't be mapped to an answer type. This is however not considered as a problem within the ATD module, because it's the lexicon maintenance that's responsible to provide a good and complete set of mapped phrases.

5.4.3 Semantic Phrase Detection (SPD)

The goal of the SPD component is to consider all the phrases returned by the phrasing component, and check if there exists at least one semantic item in the lexicon of phrases that relates to it. Basically, this means that the SPD component relies on the phrasing performance and the accuracy of the lexicons.

The evaluation results of 70% is mostly due to missing references in the phrase-lexicon, and in some cases because of bad chunks. This means that there is not really much improvements to do with this component alone, but rather improve the lexicons. Errors that were caused due to bad chunks could in theory be fixed by ignoring the returned phrases from step 1, and in stead have the SPD component search through the entire question for semantic phrases. However, this approach disregards the design of the program.

A task that could be considered for future work is to implement a similarity function in stead of looking for complete matches. For example, if the lexicon only includes the term "winner" and not the plural form "winners", the phrase "winners" should also be considered mapped to the same semantic item as "winner", because it would yield a high similarity score.

5.4.4 Semantic Phrase Mapping (SPM)

The SPM component is closely connected to SPD, if a phrase cannot be detected then it cannot be mapped correctly either. SPM also reflects the lack of statistical terms. Phrases like "most points", "more than ten", "since 2010" etc. are not yet implemented in the question processing module. Because statistical terms are not necessarily domain-dependent, it's debatable if these phrases should be included in the lexicon of phrases. Perhaps it would be better to use another set that includes statistical terms, especially considering future expansion of the program which could include multiple sport domains.

However, statistical terms are of high importance to the program and should be the top-priority of improvement tasks in the future.

5.4.5 Named Entity Recognition (NER)

NER is similiar to SPD and also relies on the phrasing in step 1, in addition to the lexicon of resources. The overall results of NER (80%) is affected by false positives, and the improvement point for this component is to remove all the false positives which may influence the query generation.

What's good about the NER component is that it recognised all the NEs from the baseline questions, despite dealing with multiple poorly-chunked phrases, and in the end yielded a high recall score.

5.4.6 Named Entity Disambiguation (NED)

Apart from concepts, which are recognised but incorrectly mapped, NED are performing quite good and correctly maps all the *relevant* NEs. However, just like with NER there are a few false positives that affects the results. This is partly because of bad chunks which includes too many or too few words in one phrase. For example, from the baseline question CQ34 (see Appendix A) the phrase "average points Lewis Hamilton" is chunked as one phrase (because of consecutive nouns). In this particular example, NED correctly maps it to the driver Lewis Hamilton, but the similarity score is heavily reduced.

The main challenge of this component is the similarity function that compares two strings, which definitely needs an improvement. The similarity score is calculated by taking the average of three different approaches, but there is no good reason for why these three were selected and used. When the function was developed, the idea was to experiment with different approaches and select the best performing method. There isn't a formal representation of these experiments presented in this paper, but the initial results indicated that a combination of all the three methods yielded the best result. However, due to time limitations on this thesis there were other tasks that had higher priority. Even though the function is not close to perfect, it was considered as a good attempt for a first version, also represented by the high recall score.

Worth mentioning, NED is also responsible of mapping a candidate to *the correct entity*. For example, the phrase "Schumacher" can refer to both Michael

(MS) and Ralf Schumacher (RS). However, none of the baseline questions included such a challenge. This was partly on purpose, because one of the assumptions as described earlier was that users are expected to refer to NEs with e.g. their full name. Otherwise, future work should also include a disambiguation process that not only depends on a similarity function, but also takes into account which entity it should be mapped to based on the other data retrieved from the question. In the Schumacher example above, Ralf would yield a better similarity score than Michael (using the current function), simply because the word "Ralf" is shorter than "Michael" and hence has less "incorrect characters". However, MS is a more successful and popular driver than his brother Ralf, and it's safe to assume that most questions involving the word "Schumacher" refers to him.

5.4.7 Query Generation (QG)

The final part of the pipeline consists of constructing SPARQL queries based on the data collected of the other components. Mapped semantic phrases are treated as predicates, and mapped resources are treated as subjects or objects in the query. The answer type, if determined, also affects the procedure. The QG component, especially the default handler of the function, is referred to as a "greedy" approach because it tries to create as many queries as possible, "hoping" that at least one of them will be correct. Most of the competency questions are handled by the default handler, and even though this is a relative simple-built function, it manages to answer correctly on a good portion of these questions.

The special handlers implemented for different cases works quite good too, and every question out of the 50 baseline questions that are meant to be handled by one of these functions do so correctly. However, using custom built functions like this does not work in the long run. Expanding the domain would require even more manually created functions, and hence the component is not very adaptable for changes.

The QG component works best for those questions that seek a simple fact. Questions that go beyond these cases looking for more complex relations, becomes a challenge for the QG. For example, question CQ25: *Name a circuit in Spain*, should in theory be easily converted into a SPARQL query, especially since both the terms "circuit" and "Spain" are correctly mapped to the semantic items. However, because there is no direct relation between circuits and their locations in the ontology, the QG component fails to see the connection.

To summarise, it's difficult to implement a component for generating queries based only on input data from the other components. Specifically, it's been difficult to convert the logic and the sentence structure that comes natural to humans, into a machine-readable format. The implementation presented in this paper fails to address these challenges, but rather "hopes for the best outcome" based on the available data. Through the evaluation we have shown that this approach works for simple cases/questions, but further research and development of this component should be prioritised for future work in order to

correctly answer more complex questions.

5.5 General

5.5.1 Approach

Related to previous work within the field of QA systems over Linked Data, there are many different approaches used today. Compared to this implementation, there are different methods both within each specific component but also on a higher level such as the main pipeline. For example, for phrase detection it's possible to use lexicons, third party libraries or a combination. Named entities can be looked up using the knowledge base, a NER module from a third party or even using the Web. Structure-wise, some approaches splits the question into phrases which are further processed separately, other approaches do not.

It was never a goal with this implementation to specifically try and come up with a new, improved method. It was meant to be an introduction to the field, learning about what techniques and methods that existed and try to implement a system from scratch. Of course it would be great if we could present an interesting and revolutionary approach to the field, and it would also be considered as a great bonus if this thesis could inspire others to come up with new solutions as well. With this in mind, we believe that we have succeeded with the task of implementing an answering engine using Linked Data, however we cannot call it a revolutionary approach that is new to the field.

5.5.2 Prototype

Limited by time, we did not manage to put a lot of effort into developing a prototype, even though we believe we have delivered a decent user interface that utilises the program. However, there are some features missing that we specifically wanted to include.

First of all, we wanted to include a feature that supported real-time user feedback. This means that if the question was interpreted incorrectly, the user would get the option to modify the results on component level, e.g. specifying what named entities that were incorrect or missing, or even modifying the queries directly.

Second, it would be ideal to include a list of predicates/properties associated with detected resources, so users could look up more information. For example, if the user asks a question about the team McLaren, then provide a list of other properties (from the ontology) related to that team.

Finally, the web interface does not clearly present the underlying data that was used to find the answers. In stead, we provide links for the returned resources such as the DBPedia page of the resource. Links to the Wikipedia pages could also be included.

5.5.3 Evaluation

Some additional comments on the evaluation: Many of the components rely on the performance of other components in the pipeline, and a failure in one step can later lead to a failure in another step. Examples of this phenomena has already been reviewed. For example, if NER falsely detects a phrase as a NE candidate (a false positive), the NED component will still try and map it to an entity. Because NED uses a similarity function, it may or may not return an entity. If it does, it will definitely be an incorrect mapping. We refer to these faults as the "pipeline effect", and is well demonstrated in Figure 4.7.

In this presentation we have included faults related to the pipeline effect, which means that some components may actually perform better than what's presented. The reason is that it's difficult and in some cases impossible to know how many of the pipeline steps that actually affected the results directly, and how many of the questions that were affected because of the pipeline effect. For example, consider a question that has failures in the components phrase detection, phrase mapping, NER, NED and QG. Even if the fault was corrected in phrase detection, it may still be a failure in the NED component preventing further processing of the question, and so on. In theory, it's possible to simulate each failure, correcting them manually and then end up with a more accurate component evaluation, but this is very time consuming and would probably (educated guess) not change the final performance very much. In stead, we analysed typical errors (see Section 4.2.5) related to what we believed was the root cause, like missing statistical terms and failing on addressing concepts.

5.5.4 Overall goals

We repeat the primary tasks of this assignment as first described in the introduction along with some concluding marks.

Collecting data or connecting to existing data services

We successfully managed to collect data using Wikipedia as the main source. This was carried out by developing coded scripts that automatically could retrieve Wikipedia data, but some data was collected manually. For the future, all data should be collected using an automatic approach in order to be consistent and avoid human errors like typos.

Building a conceptual model of the chosen domain of sports

We managed to build an ontology for the domain of Formula 1, including different classes, relations and properties. Some parts of the ontology could be reused from DBPedia, a Linked Data representation of Wikipedia content. Other parts had to be created using a personal repository. Expanding and improving the ontology will be a continuous task, especially if more sport domains are to be included.

Developing a question interpreter that can understand concepts related to the selected field of sports and related to statistics

A question interpreter for the domain of Formula 1 statistics was developed and could successfully detect and understand different concepts related to the sport. The program does not handle statistical terms yet, and hence we have not succeeded on this task. This should have been prioritised, but it was also necessary to learn the basics of question answering. Simple factoid and list questions were therefor considered first.

Developing an answering module that can process the interpreted question against a knowledge base of facts, and in addition to showing the answer, show the underlying data that was used for the calculations

The answering module was originally thought to be a separate module, and would get input from the question processing module in the format of SPARQL queries. However, we modified this task slightly and in stead return the answers directly to the answering module, in the form of either a literal value or a URI (or a list of them). The answering module converts URIs into labels and presents a more user-friendly result. Future work for this module could be to convert the answers into short sentences, in addition to provide more information. For example, in stead of returning the number 77, return "Michael Schumacher was the fastest driver in 77 laps".

Otherwise, we present the answers in addition to the results of each of the components, acting as evidence of the answer calculation. As mentioned, we should also have presented the information source directly in stead of linking to a web page.

Implementing a working prototype

As discussed, we have delivered a working prototype of the system that works, though there are some interesting features missing.

Performing an evaluation of the system

We performed an evaluation of the system, both in terms of end-to-end and on component level for the question processing module. The knowledge base and the ontology design were also evaluated and discussed, but these results are not presented in this paper.

Chapter 6

Conclusion

6.1 Summary

Throughout this paper we have reviewed the exciting and growing field of Linked Data and The Semantic Web. We saw how the RDF framework were utilised for creating Linked Data, and how to create linked statements about resources through RDF triples. We discussed how data could be organised with ontologies, how RDF data was stored in knowledge bases and how to retrieve data with SPARQL queries.

We introduced the concept of question answering systems and presented a brief insight to the historical evolution of these systems. We discussed the importance of having a natural language interface, so that people could retrieve relevant and precise information from structured data.

Based on the existing theory and inspired by related work, we developed a question answering system for the domain of Formula 1 statistics. We did this by first creating a conceptual model of the intended domain knowledge and collected relevant information from Wikipedia. We then developed a question interpreter that could analyse and understand different concepts related to the knowledge domain. The question processing module was able to convert a natural language question into one or more SPARQL queries, which in turn were executed on the knowledge base for answer retrieval. After adding a web interface on top of this program, we presented a working prototype of the QA system.

A full evaluation of the system was carried out, based on a set of 50 baseline questions. Through the end-to-end performance results we saw that the system could answer almost 50% of the questions, but handled easy factoid question better than complex, aggregated questions. The performance of the components revealed that the system is affected by the pipeline effect, where a previous step in the pipeline directly affects the performance of the next step. The error analysis showed that the lack of addressing statistical terms is the greatest weakness of the system.

6.2 Future work

There are still a lot of work needed in order to make the program more robust and adaptable for future expansion. We summarise some of the most important steps that should be considered as improvement points for future work.

- Improve the phrasing in order to increase the accuracy of semantically-correct chunks.
- Improve or replace the existing similarity function used for NED, which could also be applied to NER. Resource lexicons combined with online services or third party libraries is a suggestion.
- Automatic maintenance of lexicons in addition to include a wider spectrum of synonyms (synsets). An example is to use WordNet.
- More research is needed in order to improve query generation, in means of understanding the semantic structure of the sentence and the logical build-up of the query. Specifically, improve the creation of complex queries that includes aggregation features.
- Further develop the user interface regarding answer preparation and presentation. The underlying data that was used as evidence for "calculating the answer" should be displayed.
- Expand the ontology and the knowledge base to support other sport domains. Connecting to a live service for collecting data could be interesting.

Bibliography

[1] Andrejs Abele et al. *Linking Open Data cloud diagram 2017*. [Online; accessed 04-June-2017]. URL: http://lod-cloud.net/.

[2] Hannah Bast, Björn Buchhold, and Elmar Haussmann. “Semantic Search on Text and Knowledge Bases”. In: *Foundations and Trends® in Information Retrieval* 10.2-3 (2016), pp. 119–271.

[3] Steven Bird, Ewan Klein, and Edward Loper. “Natural Language Processing with Python”. In: *O'Reilly Media* 1st edition (2009), ch.5, ch. 7.

[4] C. Bizer, T. Heath, and T. Berners-Lee. “Linked data - the story so far”. In: *Int. J. Semantic Web Inf. Syst.* 5.3 (2009), pp. 1–22.

[5] Blank node. *Blank node — Wikipedia, The Free Encyclopedia*. [Online; accessed 06-June-2017]. URL: https://en.wikipedia.org/wiki/Blank_node.

[6] Confusion Matrix. *Confusion Matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 08-June-2017]. URL: https://en.wikipedia.org/wiki/Confusion_matrix.

[7] Formula One Constructors. *Formula One Constructors — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-May-2017]. URL: https://en.wikipedia.org/wiki/List_of_Formula_One_constructors.

[8] Hoa Trang Dang, Jimmy Lin, and Diane Kelly. “Overview of the TREC 2006 question answering track”. In: *Proceedings of the Fifteenth Text REtrieval Conference, TREC 2006, Gaithersburg, Maryland, USA, November 14-17, 2006* (2006), pp. 99–116.

[9] DBPedia. *About DBpedia*. [Online; accessed 19-April-2017]. URL: http://wiki.dbpedia.org/about.

[10] Factoid. *Factoid — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-April-2017]. URL: https://en.wikipedia.org/wiki/Factoid.

[11] Anthony Fader et al. *ReVerb*. [Online; accessed 12-June-2017]. URL: http://reverb.cs.washington.edu/.

[12] Formula One. *Formula One — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-May-2017]. URL: https://en.wikipedia.org/wiki/Formula_One.

[13] Bert F. Green Jr. et al. "Baseball: An Automatic Question-answerer". In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '61 (Western). Los Angeles, California: ACM, 1961, pp. 219–224. DOI: 10.1145/1460690.1460714. URL: http://doi.acm.org/10.1145/1460690.1460714.

[14] Stanford NLP Group. *Stanford Log-linear Part-Of-Speech Tagger*. [Online; accessed 13-March-2017]. URL: http://nlp.stanford.edu/software/tagger.shtml.

[15] LinkedDataTools.com. *Linked Data Tools Tutorial*. [Online; accessed 04-June-2017]. URL: http://www.linkeddatatools.com/introducing-rdfs-owl.

[16] Merriam-Webster. *Factoid definition*. [Online; accessed 12-April-2017]. URL: https://www.merriam-webster.com/dictionary/factoid.

[17] NIST. *2004 TREC-QA Relationship Questions summary*. [Online; accessed 12-April-2017]. URL: http://trec.nist.gov/data/qa/add_QAresources/README.relationship.txt.

[18] NIST. *Question Answering Collections*. [Online; accessed 14-February-2017]. 2005. URL: http://trec.nist.gov/data/qa.html.

[19] NIST. *TREC Overview*. [Online; accessed 14-February-2017]. 2014. URL: http://trec.nist.gov/overview.html.

[20] NIST. *TREC Tracks*. [Online; accessed 14-February-2017]. 2017. URL: http://trec.nist.gov/tracks.html.

[21] NLTK. *NLTK 3.0 Documentation - nltk.tokenize*. [Online; accessed 14-April-2017]. URL: http://www.nltk.org/api/nltk.tokenize.html.

[22] Natalya F. Noy and Deborah L. McGuinness. *Ontology Development 101: A guide to Creating your first ontology*. Tech. rep. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880. 2001.

[23] John Prager. "Open-domain Question: Answering". In: *Found. Trends Inf. Retr.* 1.2 (Jan. 2006), pp. 91–231. ISSN: 1554-0669. DOI: 10.1561/1500000001. URL: http://dx.doi.org/10.1561/1500000001.

[24] RDF Query Language. *RDF Query Language — Wikipedia, The Free Encyclopedia*. [Online; accessed 05-June-2017]. URL: https://en.wikipedia.org/wiki/RDF_query_language.

[25] Resource Description Framework. *Resource Description Framework — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-February-2017]. 2001. URL: https://en.wikipedia.org/wiki/Resource_Description_Framework.

[26] Beatrice Santorini. "Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision)". In: *University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-47.* (1990), p. 6.

[27] Scholarpedia. *Word sens disambiguation.* [Online; accessed 23-February-2017]. URL: http://www.scholarpedia.org/article/Word_sense_disambiguation.

[28] Text Retrieval Conference. *Text Retrieval Conference — Wikipedia, The Free Encyclopedia.* [Online; accessed 14-February-2017]. 2005. URL: https://en.wikipedia.org/wiki/Text_Retrieval_Conference.

[29] Linked Data Community Tom Heath. *Linked Data - Connect Distributed Data across the Web.* URL: http://linkeddata.org/home.

[30] Christina Unger and Philipp Cimiano. "Pythia: Compositional Meaning Construction for Ontology-Based Question Answering on the Semantic Web". In: *Natural Language Processing and Information Systems - 16th International Conference on Applications of Natural Language to Information Systems, NLDB 2011, Alicante, Spain, June 28-30, 2011. Proceedings.* 2011, pp. 153–160.

[31] Christina Unger, André Freitas, and Philipp Cimiano. "An Introduction to Question Answering over Linked Data". In: *Reasoning Web. Reasoning on the Web in the Big Data Era: 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings.* Ed. by Manolis Koubarakis et al. Cham: Springer International Publishing, 2014. ISBN: 978-3-319-10587-1. DOI: 10.1007/978-3-319-10587-1_2. URL: http://dx.doi.org/10.1007/978-3-319-10587-1_2.

[32] Christina Unger et al. "Template-based Question Answering over RDF Data". In: *Proceedings of the 21st International Conference on World Wide Web.* WWW '12. Lyon, France: ACM, 2012. ISBN: 978-1-4503-1229-5. DOI: 10.1145/2187836.2187923. URL: http://doi.acm.org/10.1145/2187836.2187923.

[33] W3C. *Linked Data.* [Online; accessed 23-February-2017]. URL: https://www.w3.org/standards/semanticweb/data.

[34] W3C. *Resource Description Framework (RDF).* [Online; accessed 03-March-2017]. URL: https://www.w3.org/RDF/.

[35] W3C. *Semantic Web.* [Online; accessed 23-February-2017]. URL: https://www.w3.org/standards/semanticweb/.

[36] W3C. *Tim Berners-Lee.* [Online; accessed 03-March-2017]. URL: https://www.w3.org/People/Berners-Lee/.

[37] W3C. *Web Ontology language (OWL).* [Online; accessed 04-June-2017]. URL: https://www.w3.org/OWL/.

[38] Wolfram Alpha. *Wolfram Alpha — Wikipedia, The Free Encyclopedia.* [Online; accessed 06-June-2017]. URL: https://en.wikipedia.org/wiki/Wolfram_Alpha.

[39] W A Woods. "Readings in Natural Language Processing". In: ed. by Barbara J. Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber. Morgan Kaufmann Publishers Inc., 1986. Chap. Semantics and Quantification in Natural Language Question Answering, pp. 205–248. ISBN: 0-934613-11-7. URL: `http://dl.acm.org/citation.cfm?id=21922.24336`.

[40] WordNet. *WordNet — Wikipedia, The Free Encyclopedia.* [Online; accessed 07-June-2017]. URL: `https://en.wikipedia.org/wiki/WordNet`.

[41] Princeton University WordNet. *WordNet.* [Online; accessed 26-April-2017]. URL: `https://wordnet.princeton.edu/`.

[42] YAGO. *YAGO — Wikipedia, The Free Encyclopedia.* [Online; accessed 19-April-2017]. URL: `https://en.wikipedia.org/wiki/YAGO_(database)`.

[43] Mohamed Yahya et al. "Natural Language Questions for the Web of Data". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning.* EMNLP-CoNLL '12. Jeju Island, Korea: Association for Computational Linguistics, 2012, pp. 379–390. URL: `http://dl.acm.org/citation.cfm?id=2390948.2390995`.

Appendix A

Baseline Questions

The complete set of the 50 baseline questions.

CQ.N	Competency Question	Type	Category
CQ01	Who won the Australian Grand Prix in 2011?	1	FACTOID
CQ02	When was the start date of Canadian Grand Prix in 2012?	1	FACTOID
CQ03	Which team won the German Grand Prix in 2014?	1	FACTOID
CQ04	What was the course length of the Japanese Grand Prix in 2010?	1	FACTOID
CQ05	What was the circuit of the Malaysian Grand Prix in the 2011 season?	1	FACTOID
CQ06	Who was the pole driver of the Hungarian Grand Prix in 2010?	1	FACTOID
CQ07	Which team became champion in the 2012 season?	1	FACTOID
CQ08	Which Grand Prix was Jenson Button's first win?	1	FACTOID
CQ09	How many F1 races has McLaren participated?	1	FACTOID
CQ10	How many fastest laps was recorded for Michael Schumacher?	1	FACTOID
CQ11	List the teams that participated in the 2015 season	1	LIST
CQ12	How many times has Sebastian Vettel been in the top three?	1	FACTOID
CQ13	How many times have McLaren drivers won the World Championship?	1	FACTOID

CQ14	How many times have Redbull Racing won a Grand Prix?	1	FACTOID
CQ15	Which Grand Prix did Red Bull Racing win for the first time?	1	FACTOID
CQ16	Which country did the 2014 season start in?	1	FACTOID
CQ17	Which driver became champion in the 2012 season?	1	FACTOID
CQ18	Which driver came in third place in the 2014 Italian Grand Prix?	1	FACTOID
CQ19	List all the winners in the 2011 season	1	LIST
CQ20	How many points did Lewis Hamilton get in the 2015 season?	2	FACTOID
CQ21	Which drivers had the pole positions in the 2011 Season	2	LIST
CQ22	List the teams that Fernando Alonso drove for since 2005?	2	LIST
CQ23	Which Grand Prix was Jenson Button the fast driver?	1	LIST
CQ24	What was the most points a driver got in the 2015 season?	2	FACTOID
CQ25	Name a circuit in Spain	2	LIST
CQ26	Who are the top three drivers with the most podiums in a single season?	2	LIST
CQ27	In what races was Daniel Ricciardo the fastest driver?	1	LIST
CQ28	At which position did Jenson Button end the 2014 season?	2	FACTOID
CQ29	How many wins had Sebastian Vettel in the 2011 season?	2	FACTOID
CQ30	List all the german drivers in the 2015 season	2	LIST
CQ31	List drivers with more than 90 podium positions	2	LIST
CQ32	Which team has won the most Grand Prix since 2010?	3	FACTOID
CQ33	Which countries has hosted the least amount of Grand Prix?	3	LIST
CQ34	What is the average points Lewis Hamilton gained in all seasons?	3	FACTOID
CQ35	List all the drivers who has won the Autralian Grand Prix more than once	3	LIST

CQ36	Which team has won the most Grand Prix in Spain?	3	FACTOID
CQ37	List all the teams who have won the Italian Grand Prix more than once	3	LIST
CQ38	What is the winning percentage for Michael Schumacher?	3	FACTOID
CQ39	Which season had the longest race distance in total?	3	FACTOID
CQ40	Which driver had the most consecutive pole positions since 2010?	4	FACTOID
CQ41	Who is the youngest driver to become a champion?	4	FACTOID
CQ42	Who is the youngest driver with more than one win?	4	FACTOID
CQ43	How many points did Nico Rosberg get in the Canadian Grand Prix in 2014?	4	FACTOID
CQ44	Who was the winner of the 2006 season?	1	FACTOID
CQ45	Which race was the shortest?	2	FACTOID
CQ46	Which season had the most races?	2	FACTOID
CQ47	Which team has the most podiums in the history of F1?	3	FACTOID
CQ48	How many times did Kimi Raikkonen win with McLaren?	3	FACTOID
CQ49	Who has more fastest laps than Niki Lauda?	2	LIST
CQ50	What team has most wins?	2	LIST

Appendix B

Overview of Attachments

The following files are included as attachments (zipped file).

- *code.py* - The complete Python source code of the question processing module.
- *lexicons.py* - Source code of Python script that is used to build the lexicons.
- *seasonDrivers.py* - A Python script used for collecting data from Wikipedia/DBpedia. Was further developed by Pirbazari.
- *kb-resources.tsv* - The lexicon of resources
- *phrases.tsv* - The lexicon of phrases