UiO : Department of Informatics
University of Oslo

# OptiqueNLQF: A natural language query formulation system based on Semantic Technologies

Tomas Stein Sæbu

Master's Thesis Autumn 2015

# OptiqueNLQF: A natural language query formulation system based on Semantic Technologies

Tomas Stein Sæbu

3rd August 2015

# Acknowledgements

First of all I would like to thank my supervisors Martin Giese, Jan Tore Lønning and Dumitru Roman for help and guidance along the way.

Secondly I want to thank my family and friends for their continuous positive support.

A special thanks to Carl Martin, Lars Kristian, Jonas, Kristian, Trond, Petter and Helene for all the hours of discussion and supporting conversations.

Thank you all!

# Abstract

This master project investigates the possibilities of having a natural language query formulation system where users will be able to specify an initial version of a query. Instead of confining users to pointing, clicking, scrolling and searching the menus for the right classes and attributes of a visual interface, we want to provide the opportunity to express their information requests in their own words. We want to create a system that is usable for a wider range of people, and where the technical background and knowledge about query generation no longer needs to be a requirement. Making database searches available for the average worker will increase the information flow, when everyone can retrieve the information they are interested in without having to ask specialists, non-technical staff are free to focus on their primary tasks and the information flow is undisrupted.

The OptiqueNLQF-system analyzes an input information request and generates a SPARQL query. The prototype developed in this thesis is capable of handling simple natural language input phrases and will generate a functioning SPARQL query. Throughout this thesis we will discuss the design choices which have been made and the implementation of the system, as well as what possible improvement can be made.

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction

# Chapter 1

# Background

## 1.1 Introduction

With the colossal amount of data stored in enterprises increasing, the continuous need to retrieve this data faster and easier is a never ending struggle for computer scientists. More aspects of business are driven by data and more people are depending on quick data access. This results in a need for people with little knowledge of query extraction to be able to attain the data they need themselves.

The overall goal of this thesis is to implement a prototype of a natural language query formulation system. The idea is for a user to formulate a phrase, an information request, and be able to retrieve data through a query execution on a database. This prototype will be implemented as a component of the Optique platform.

## 1.2 Optique

Optique [1] is an EU funded project coordinated by the University of Oslo. It is a scalable end-user system for access to big data. Optique aims to provide a semantic end-to-end connection between users and data sources, and in this way make it easier for people with limited programming and query language experience to search for, and retrieve data. The Optique system has a menu based "point and click" navigation module, a "visual query formulation" tool, OptiqueVQS [17]. This is illustrated with a screenshot from the working system in Fig. 1.1. The user can choose between the ontology concepts in a table, select relations or attributes (e.g. names, values) and create a graph describing the information needed in the visual interface. The OptiqueVQS then formulates a SPARQL query and executes the query on the database.

[1] http://http://optique-project.eu/

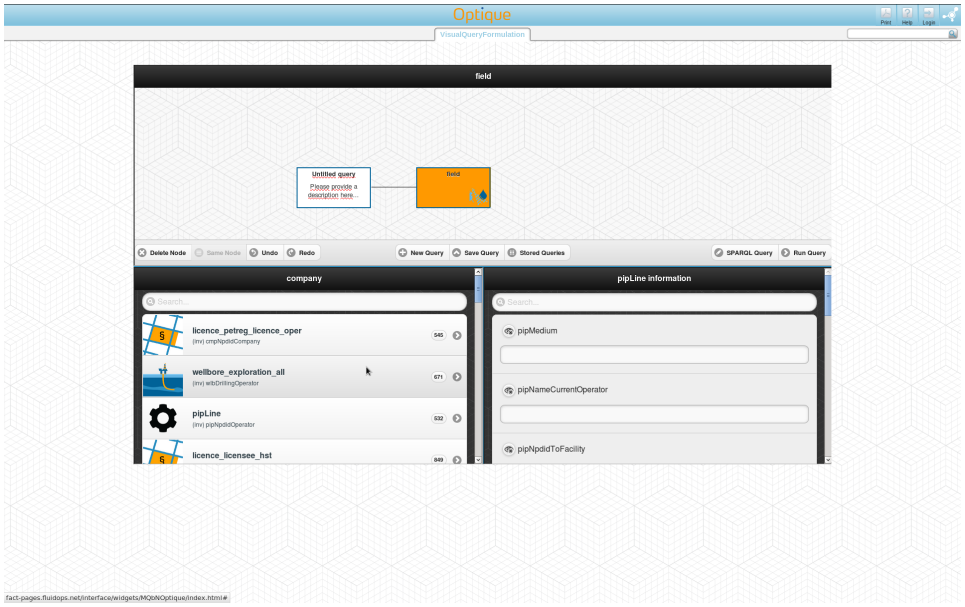



Figure 1.1: Screenshot of Optique interface.

## 1.3 Goals

This master project investigates the possibilities of having a natural language query formulation system where users will be able to specify an initial version of a query. Instead of confining users to pointing, clicking, scrolling and searching the menus for the right classes and attributes, we want to provide the opportunity to express their information requests in their own words. We want to create a system that is usable for a wider range of people, and where the technical background and knowledge about query generation no longer needs to be a requirement. Making database searches available for the average worker will increase the information flow, when everyone can retrieve the information they are interested in without having to ask specialists, non-technical staff are free to focus on their primary tasks and the information flow is undisrupted.

The system would work like this. A user needs some data from the database. Regardless of how complex the request is, the user formulates an initial overall part of the question. Consider the complex query in Tab. 1.3.

> *"In my area of interest (AOI) return all wellbores that penetrates the cronostrat unit <C1> and return information about the lithostratigraphy and the hydrocarbon content (saturated and moveable + shows) in the wellbore interval that penetrates the <C1> unit. Also return information about other wellbore intervals with hydrocarbon content (saturated and moveable + shows) in the wellbores with hydrocarbon in <C1>."*

> *"In my area of interest return all wellbores that penetrates the "Miocene" cronostrat unit and return information about the lithostratigraphy and the hydrocarbon content in the wellbore interval that penetrates the "Miocene" unit. Also return information about other wellbore intervals with hydrocarbon content in the wellbores with hydrocarbon in "Miocene"."*

Note that <c1> in the first version of the quote is a variable representing an attribute, e.g. "Miocene", AOI is "area of interest". The initial query would be *"return all wellbores that penetrates the cronostrat unit"*. The user will then be presented with a graph representation of the question, on the form shown in Fig. 1.1. Now the user will have the opportunity to add more specifications to the query by adding attributes to the graph in the menu-system, or by writing more specific information request. The system will then again update the graph, and the user will be able to change or add to the graph. When the user is satisfied, he runs the query. The user will now be presented with the lists and tables containing the requested information.

# Chapter 2

# Previous work

## 2.1 State of the Art

Beginning a project like this, it is important to look into which methods and approaches have been used before, to get some pointers on how we are going to tackle the problem at hand. Therefore we investigate how others have handled these problems in similar projects, and identify what has worked for them and consider if what they have done is relevant for us. The following are projects with the same general goal as ours, a natural language query formulation/question answering system, which we have looked closer into.

### 2.1.1 SINA

The SINA project [15] is developed at the University of Leipzig [1]. It is a natural language question answering system where they emphasize the challenges of working with data from different data sets, and they want to exploit the links between the data sets. To do this they match segments, a piece of a sentence or in this case a word, of the input query against all available data sets. With these segments they create $N$-tuples, and again create triples from these segments and map the triples together to make query graphs. The system employ a Hidden Markov Model [2] (HMM), which is a statistical method for finding the optimal segments from the input. A HMM is used by performing supervised learning on a training set with the similar language to what will be used in the system. By doing this the program learns what the transition probabilities and emission probabilities are. Transition probability is the probability that one state is followed by one other state. In language tagging a state is usually a part-of-speech (POS) tag. Emission probability is the probability that a given observation emits from a state. An observation is in this case a word from the input query. The input query is a set of states, and we know what POS-tags the states most likely are tagged as, we also know what POS-tags most likely are following every POS-tag. The Viterbi Algorithm [9] is now applied

[1] https://www.zv.uni-leipzig.de/en/

[2] http://en.wikipedia.org/wiki/Hidden_Markov_model

to calculate the optimal path. After this is done the query is correctly disambiguated and segmented. After the input is segmented it is time to construct the query graph. To construct a query graph one starts out by making an incomplete query graph containing all possible combinations of the resources detected from the query input. This will produce two or more disjoint sub-graphs. To connect these sub-graphs a minimal spanning tree method is used. The approach used for connecting the graphs by SINA are Prim's Algorithm. Prim's algorithm is an algorithm for finding the minimum spanning tree from a graph, like the one earlier created. To connect disjoint graphs the *owl:sameAs* links are applied to compute a comprehensive set of properties.

This system has many similarities to what Optique wants with their natural language system, but it may not be as complex as Optique aims to be. The way SINA removes stop-words and uses lemmatization to create $N$-tuples of the input query can remove part of the meaning from some sentences. And by this make the system less robust when dealing with plain text input, and well structured sentences formulated in a uniform manner is not something we can rely on from our future users. Parsing sentences this way is easier, and it is a straightforward way of handling easy and well structured sentences. The way the SINA system maps segments to triples and by this making it into SPARQL queries is smart and simple and works very well when working with RDF data. It is an idea to make an initial system like this, but it would not understand the sentences we have been given as example queries. We have some data that can be used as training data, which has proven to be one thing that is a challenge when working with classifiers, which also has the potential to be very expensive. This training data makes a good starting-point if we decide to use the HMM method. The Viterbi algorithm is also a fast way of finding the most likely sequence, and is something we will consider when choosing disambiguation methods.

### 2.1.2 Unified Service Intelligence

"USI answers" is a system developed by Siemens /footnotehttp://www.siemens.com/. In the paper [20] the system's method for parsing and analyzing the input is described as a work-flow pipeline system, shown in Fig **??**, with a primary and secondary knowledge base. This describes a complex system with many different components that each play its part in creating a query from the natural language input. The USI knowledge base is represented by an ontology. Here they have provided a short description of each concept and a large set of the most common synonyms have been added. The primary knowledge base is where the actual data is stored. This consists of both structured data, with clear key-value association, and unstructured data, e.g text extracted from pdf reports. The secondary knowledge base is used as a resources for additional evidence for interpretation hypothesis, and for additional potential answers. This is non-specific Siemens data, like open domain-based resources such as *DBpedia, FreeBase, GeoNames*.

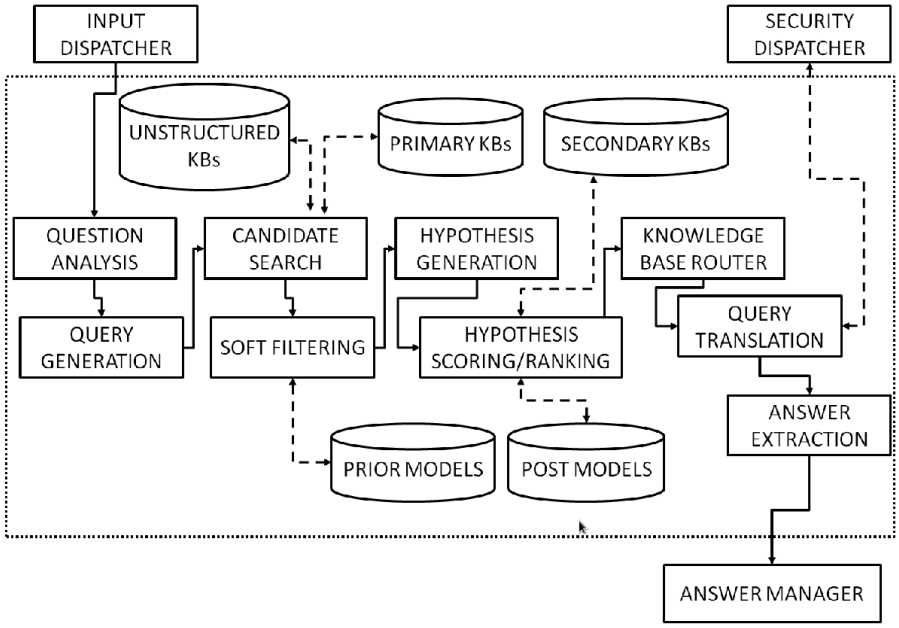



Figure 2.1: The USI work-flow pipeline.

- The pipeline from input to answer includes **question analysis**, the step where the input question is parsed and normalized. This step applies:
  - *question normalization*, which is analyzing and dealing with brackets and quotes in the input string.
  - *Metadata annotation* for adding metadata, like user keys and session keys.
  - *Question parsing* as standard parsing like part-of-speech tagging, lemmatization, disambiguation and named entity recognition.
  - *Analysis validation* for handling domain-specific input terms, different numbers and codes, after this was split and parsed in the parsing.
  - *Question classification* is analyzing what type the question is (e.g. fact-based or list-based), how the answer will be represented and the question focus.
- The second step is **query generation** where the input question is transformed into a query. From what was learned in the question analysis step the most likely query type and the most likely query format is generated. This component supports Dictionary- and Regular Expression look-ups, Apache Lucene-, SPARQL-, and SQL-based queries.

- **Candidate search** is a step where the goal is to search for and categorize concepts (called answerFields), concept value instances (called searchFields) and augmented key-value pairs (called domainFields).

- The next step in the pipeline is **soft filtering**. Here the system's detects and validates the categorization from the candidate search, and rank the different annotation referenced to a query token. This is done with pre-learned prior models.

- **Hypothesis generation** is a step where different interpretations of the analysis is generated. With what was learned in the candidate search step, the different ways of how the answer field and the search field is connected are generated.

- Next come the **hypothesis scoring and ranking**. Here the hypothesizes, generated in the previous step, are scored and ranked, before the highest ranked interpretation is sent to the next step.

- The **knowledge base router** detects the data sources needed from the knowledge base for joining and querying.

- **Query translation** is where the query is constructed. The system automatically generates *SQL, Apache Lucene SPARQL and solution object* queries.

- The last step is **answer extraction**. This is where the actual answer to the input query is given.

- There is also a final step **answer manager**, this step coordinates the front-end - back-end communication. The system has an interface with several attributes to help form the information request the right way. It provides a list of concept-instance relations, so the user may change or adjust the input question. It also gives you the opportunity to add different data sources to the question, and finally you are able to give feedback on the question interpretation.

This system is the one most similar to what we are working to achieve, the way the question input is thoroughly analyzed and interpreted is something to strive for. The system interface with opportunity to change the question, and the system showing what you will be searching for is something we deem important in our project as well. In the SINA project they discussed the exact methods they were using, but in the USI project they write more conceptual, and do not go into detail on specific methods. USI is also a commercial, very big, project. And even though we can look to it for inspiration, we must keep in mind that it has taken Siemens several years and a lot of money and manpower to develop the USI answering system.

### 2.1.3 Pythia

The Pythia system [19] is an ontology-based question answering system developed at the Biefeld University [3]. In this project, natural language is transformed into a formal query, using a linguistic analysis driven by an ontology-based grammar. The grammar used to interpret the input is divided into two parts, an ontology-specific part and an ontology-independent part. The ontology-specific part contains lexical entries on individuals, concepts and properties of the ontology. This is generated automatically from an ontology-lexicon model. A framework called LexInfo[4] is used to supply the ontology with information about its verbalization. The framework creates a declarative specification of the ontology by creating a connection from concepts of the ontology to information about their linguistic realization, i.e. Part-of-speech, morphology decomposition and subcategorization frames. A mechanism for generating grammar entries, i.e. pairs of syntactic and semantic representations, are then used on the lexical entries. These linguistic representations are used to parse and interpret the natural language question input. Now the system can map the input into formal queries. This involves three main steps. First the input is parsed by constructing a linguistic tree-adjoining grammar (LTAG) derivation tree, witch only considers the syntactic part of the grammar entries. After this a derived tree and an according dependency-based underspecified discourse representation structure (DUDES) is applied using semantic and syntactic composition rules. Lastly, when all argument slots are filled, the result are a set of disambiguated discourse representation structures (DRS), and these are translated into a formal query.

There are many similarities between the Pythia project and what Optique are interested in making. The fact that they are specifically describing a system using an ontology model, and there are mentioned several interfaces and structures that are relevant for Optique. Lexinfo, as a model to represent lexical information to ontologies, is a model that we have to take into consideration when dealing with the disambiguation. This project also provides the LTAG model for generating domain-specific grammars. This is also something that is of great interest to Optique.

### 2.1.4 Quepy

Quepy [5] is a framework for transforming natural language questions into database queries. It is developed by a company called Machinalis. The query transformation is done by first applying a regular expression method, where the regular expressions are predefined to match the natural language questions. The input question is parsed using a library called REfO (Regular Expressions of Objects), which is also developed by Machinalis [6], that uses tokenization and lemmatization on the words. After this the

[3] http://www.uni-bielefeld.de/
[4] lexinfo.net
[5] http://quepy.machinalis.com/
[6] http://www.machinalis.com/

system use the NLTK interface to part-of-speech tag the tokens in the input question. Natural Language ToolKit (NLTK) is an platform for Python that handles several natural language processing issues, in this case part-of-speech tagging. Quepy then constructs the tokens into triples in adjacency lists, merges the triples, and prints them out as a SPARQL-query (Quepy also has the option to make MQL-queries).

The Quepy project is a commercially built natural language to query interface, it has a working test demo online where you can translate questions about counties and movie-stars into queries. This is a system that works perfectly within a controlled sentence structure. However, their use of regexes, and setting up a specific way for a question to be formulated, is not an ideal way to handle the huge variation of questions the Optique system is required to handle. The way this system is built works more like a "fill in the blanks" way of question answering, and is not robust. The use of NLTK is something to look into, as a easy and user frendly method to use and implement.

## 2.2 System differences

So far we have looked into what these systems contain, and how they are put together. Now we will look more into what sets them apart. This will be done by focusing on some points that must be taken into account when developing a system like this. We have created a four-point list of importent factors when developing a system:

- Setting up a new project - This is how easy implementable the system is, and how difficult it would be for someone new to put the system to use.
- Use on new domain, new vocabulary - Measures how much work is needed before the system can be used on the domain of a new user. This is where generic system will get a high score.
- Scalability - The ability for a system to handle a growing amount of sentences, be able to parse a larger variety of input questions. This is a measurement of how the system can handle increasingly complex sentences.
- Upper limit - What can the system manage? Measurement of what the system can handle, this represents the top of the systems capability.

Tab. **??** also has the open source field. This is a "is or is not" information on the systems, as it is interesting to know which ones are open source. This table is a visual representation comparing the systems.

Looking at the projects from these four points it is more the systems capability and how the system perform we are interested in, rather than specific methods used. And we can see how different approaches give different results in way of performance. If we look at the Quepy framework, where regular expressions are used and the form of all question possibilities

Table 2.1: Projects

| | SINA | USI | Pythia | Quepy |
|---|---|---|---|---|
| Setting up a new project | ✗ | ✗✗ | ✗ | ✓✓ |
| Use on new domain, new vocabulary | ✓ | ✗ | ✓ | ✓ |
| Scalability | ✓ | ✓ | ✓✓ | ✗✗ |
| Upper limit | ✓ | ✓✓ | ✓ | ✗ |
| Open source | ? | ✗ | ? | ✓ |

Table 1 shows a visualization of the differences of the systems. We have used a grading-system with the scale: ✗✗,✗, ✓, ✓✓.

are predefined in the system, scalability will have a lower score, since one would have to add more regular expressions manually. On the other hand, the system will be less complex and easier to implement for new users. This is the system out of the ones that we have looked at that stands out the most from the others. Most different from the Quepy is Siemens USI Answers system, the most complex system we have looked at. This system has a very detailed language parsing component, and a huge set of synonyms connected to its ontology. This makes the system stronger on scalability compared to the Quepy, and because of its potential to grow, it gets a high score on the upper limit. The complexity of the system, its size, and the fact that is has to have synonyms added to the ontology, USI Answers score lower on the setting up the project and use on new domain.

Pythia and Sina are two more similar systems, when it comes to capability and complexity. They also have a focus on usability on new domain, which is an important factor in creating the Optique Natural Language Query Formulation System.

# Chapter 3

# Natural Language Processing

## 3.1 Overview

Natural language processing (NLP) is a field in computer science regarding interaction between human language and computers. Human language, or natural language, will in this master thesis concern the user's connection to the system, or more precisely the user's input information request. The input will be a sentence or a phrase, a phrase being a sequence of words and not a complete sentence. The NLP tasks performed in OptiqueNLQF will primarily be tokenization [1] and sentence parsing [2].

### 3.1.1 Tokenization

Tokenization is a process of segmenting a character input sequence into a list of tokens. Tokens are usually words, punctuation or numbers. The sentence "I saw a man with a telescope." will yield the tokens: "I", "saw", "a", "man", "with", "a", "telescope", ".". Another approach is to perform lemmatization on the sentence.

#### Lemmatization

Lemmatization [7, p. 80–81] is the operation of finding the root of a given word. Different words can have a common lemmas if the root of the words are the same. *Running*, *ran* and *runs* all have the same common lemma: *run*, since this is the base form of the word. The notion of lemmatization can be applied to string matching, and can prove useful if the form of the user input varies from its intended match.

[1] http://en.wikipedia.org/wiki/Tokenization_%28lexical_analysis%29
[2] http://en.wikipedia.org/wiki/Parsing

### 3.1.2 Parsing

Parsing natural language is the process of analyzing a sequence of words using the rules of a formal grammar. The word sequence is analyzed and a parse-tree [3] is generated from the sequence of words using the rules in the language model. The parse-tree shows the syntactic relations between the words and generates phrases from the different parts of the sequence. The parser also assigns the nodes in the parse-tree with word- or phrase labels. Word labels are known as part-of-speech tags (POS-tags).

#### Language model

A Language model is a statistical model composed of word sequences [7, p. 117]. It contains the formalization of the idea of word prediction where the last word of an $N$-gram is predicted by the previous ones. An $N$-gram is a token sequence of words, a 2-gram (bigram) is a two-word sequence like "I saw" or "a man", a 3-grams (trigram) is a three-word sequence, "I saw a" or "a man with". The model predicts what word most likely follows a word sequence:

$$I\ saw\ a\ man\ with\ ...$$

This word sequence is likely to by followed by "a" or "the", and probably not "I".

## 3.2 NLP Difficulties

NLP can be divided into two main areas, natural language understanding (NLU) [4] and natural language generation (NLG) [5]. NLU is enabling computers to acquire some meaning from the natural language input, NLG involves generating natural language. We will focus on the NLU part. In NLU there are many known obstacles because understanding human language is complicated. Natural language can be inconsistent and there are different words which can have the same general meaning, or similar words with different meaning according to context. A human will understand a statement with knowledge about their surroundings or using some prior knowledge, this however is more difficult for a computer.

### 3.2.1 Ambiguities

Ambiguity refers to an interpretation of a word sequence that cannot be concluded given a set of rules. Ambiguity in natural language is prevalent and when parsing a word sequence there will often be several different semantic possibilities. Consider the sentence:

$$I\ saw\ a\ man\ with\ a\ telescope.$$

[3] http://en.wikipedia.org/wiki/Parse_tree
[4] http://en.wikipedia.org/wiki/Natural_language_understanding
[5] http://en.wikipedia.org/wiki/Natural_language_generation

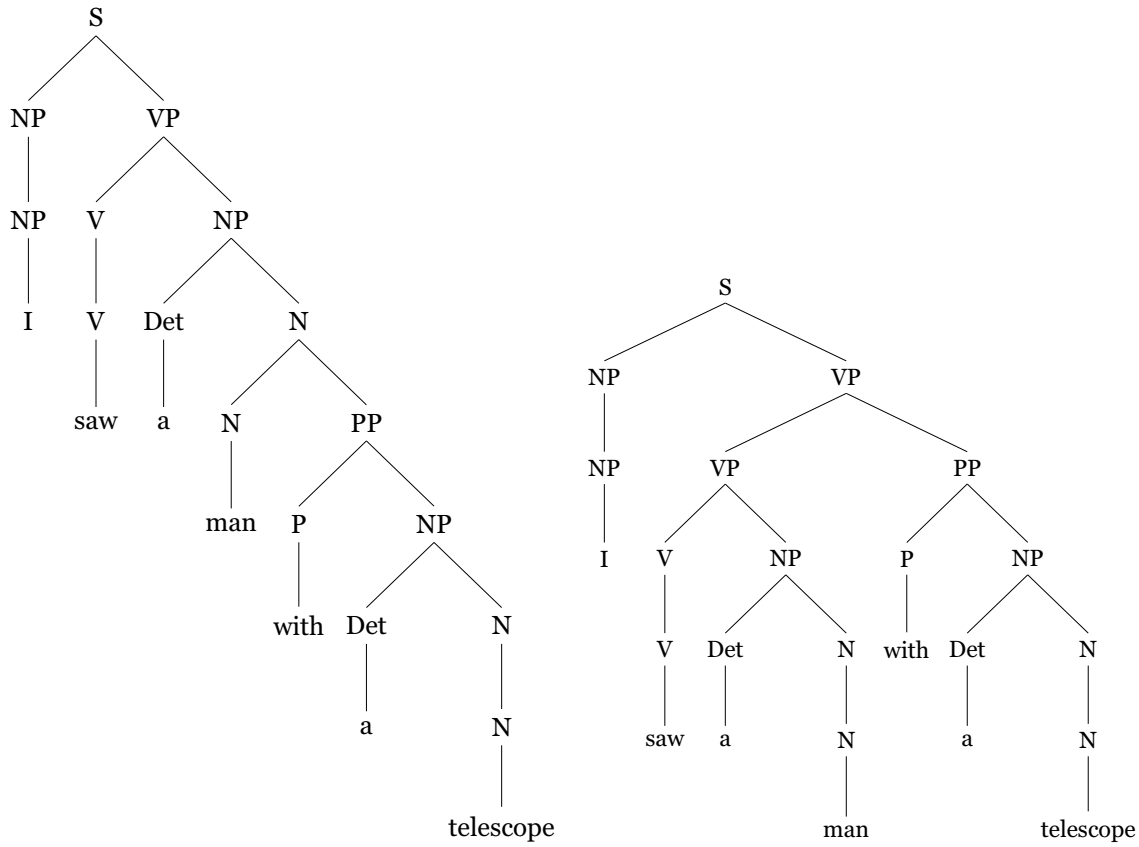



Figure 3.1: Ambiguous example sentence: parse-trees.

This is an ambiguous sentence, because it is impossible for a parser to know if the subject *I* had the telescope, or if the direct object *a man* had it. When relying on the parser to extract the needed phrases from the sentences, ambiguous sentences can prevent us from receiving our intended result. In this case it would be preferable to generate multiple parse-trees, presenting the different interpretations of the sentence.

Two different parse-tree examples from the ambiguous sentence “I saw a man with a telescope” are shown i in Fig. 3.1. We can think of every node in the tree, and the sub-tree this node produces as the phrases of the sentence. The parse-tree on the left will generate the phrases:

- saw a man with a telescope
- I saw a man with a telescope

While the parse-tree on the right will generate the following phrases:

- saw a man
- a man
- with a telescope

- a telescope
- saw a man with a telescope
- I saw a man with a telescope

Both examples will also generate the phrases representing each individual word. As we can see the rightmost parse-tree contains all the subphrases of the leftmost parse-tree, including some additional phrases. It could be preferable to use the parse-tree generating some phrases from the middle of the sentence, and not only the individual words and the complete sentence.

### 3.2.2 Conjunctions

Conjunctions often reveal problems regarding NLP in query generation systems. In the paper *Natural language interfaces to databases - an introduction* [1] its noted that the word "and" often is used to denote a dis junction rather than a conjunction, for example:

*Fields operated by Statoil and Hydro*

The meaning of this sentence is not fields that are co-operated by the two companies, rather the fields operated by either one of them. The following example demonstrates the opposite case:

*Facility type and location of statoil facilities*

In this sentence the "and" denotes that we are interested in both type and location of the facilities operated by Statoil. The mentioned paper presents the solution of calculating and presenting both alternatives in situations like this.

# Chapter 4

# Semantic Technologies

## 4.1 Semantic technologies

The definition of semantics is “the study of meaning”. Semantic technologies, as used in this project, involves storing and accessing data. For our purposes, the concepts of ontologies, resource description frameworks, web ontology languages and the SPARQL query languages are particularly important. For a general overview of the field of semantic technologies, see Foundations of semantic web technologies [6].

### 4.1.1 Ontology

An ontology [1] [14] in relation to semantic technologies is a formal description of concepts within a domain of classes and properties describing relations and attributes regarding the classes. In this project we have not created or changed any existing ontologies. It has however been important to understand and be able to extract meaning from an existing ontology.

#### Concepts/Relations/Attributes

In this thesis concepts will represent classes from the ontology, typically Field, Company and Facility. Relations are the connections between the concepts. A Field will be connected to a Company by the relation *fieldOperator* and Facility is related to Company by *pipelineOperator*. Attributes are names, values, ranges or data types connected to the concepts. This includes Field names, Company names, drilling depth or Facility type.

### 4.1.2 Resource Description Framework

The Resource Description Framework (RDF) [2] [10] is a framework for representing information. The foundation of RDF is a model for representing named properties and property values. The RDF data model consists of three object types:

[1] http://www.w3.org/TR/owl-features/
[2] http://www.w3.org/RDF/

- Resource - All things described by RDF is a resource. In our case, this involves everything described in the ontology.
- Property - A property is something used to describe a resource. It can be an attribute characteristic or relation.
- Statement - According to *Resource Description Framework (RDF) Model and Syntax Specification* [10] "A specific resource together with a named property plus the value of that property for that resource is an RDF statement". These three parts of the statement is referred to as the subject, the predicate and the object. The object in a statement can be another resource or it can be a literal. A literal is a string or some datatype connected to a resource. In the ontology we refer to literals as attributes.

A typical statement will contain a subject (resource), predicate (property) and a object (literal), and it is a representation of a statement, this is also referred to as a triple. Consider the sentence:

*Fields operated by Statoil Petroleum AS*

This sentence contains three parts:

- Subject: Field
- Predicate: operatedBy
- Object: "Statoil Petroleum AS"

### 4.1.3 Web Ontology Language

The Web Ontology Language (OWL) [13] is a knowledge language for processing content of ontologies. The most common way to write OWL syntax is OWL/RDF.

### 4.1.4 SPARQL

SPARQL Protocol And Query Language (SPARQL) [3] is a semantic query language for databases. SPARQL queries are the semantic technology-field we are most concerned with in OptiqueNLQF. A SPARQL query consists of a set of triples and optional patterns. SPARQL resembles other query languages, like SQL [4], with its use of the terms SELECT, WHERE and ORDER BY. There are several forms of SPARQL queries, CONSTRUCT, ASK, DESCRIBE, but in this thesis we will only handle the SELECT form. The difference between SPARQL and SQL is how SPARQL handles the querying. A SPARQL query is executed by matching the SPARQL triples to RDF triples. The result of a SPARQL query can be sets or RDF graphs. The triple statements in a SPARQL query resembles the RDF statements,

[3] http://www.w3.org/TR/rdf-sparql-query/
[4] http://en.wikipedia.org/wiki/SQL

except in SPARQL it is allowed to take use of variables and blank nodes. Variables and blank nodes can be in the subject or object position of the triple. Variables are used in the triples to represent something which is unknown. The predicate of the statement is a ontology relation or a identifier of what the subject is, this is represented with an *rdf:type* or simply an "a". The following is a SPARQL query representing the previous used sentence "Fields operated by Statoil Petroleum AS":

```
SELECT DISTINCT ?v1
WHERE {
?v1 a Field .
?v2 a Company .
?v1 operatedBy ?v2 .
?v2 name ?l1.
Filter(regex (?l1, ``Statoil Petroleum AS''))
}
```

Listing 4.1: SPARQL query

We can see from the query above that the variables ?v1, ?v2 and ?l1 is used. We will use ?v{number} for variables regarding concepts and ?l{number} for variables representing literals. This query express that we are interested in the fields that are operated by a company, and the company has the name "Statoil Petroleum AS". To extract data for a given name or value the pattern FILTER is used. The queries created by OptiqueNLQF will not contain more complicated query terms than this.

# Part II

# The project

# Chapter 5

# OptiqueNLQF

## 5.1 Introduction

In the following chapters we will present the OptiqueNLQF-system, and give a thorough description of how the system is put together. We will go into detail of how each part of the system works and what it produces. Each of the system's main parts will be described in its own chapter, after we have described the system architecture overview. Going into the different parts of the system we will take a close look at the the algorithm, the design choices which have been made, possible changes to the implementation and discuss some improvements that can be done in the future. After the system is thoroughly described we will evaluate it, go into how the system is tested, describe what is used during these tests and how the system performs. After the evaluation comes the conclusion and a general summary of what should be done with the system in the future.

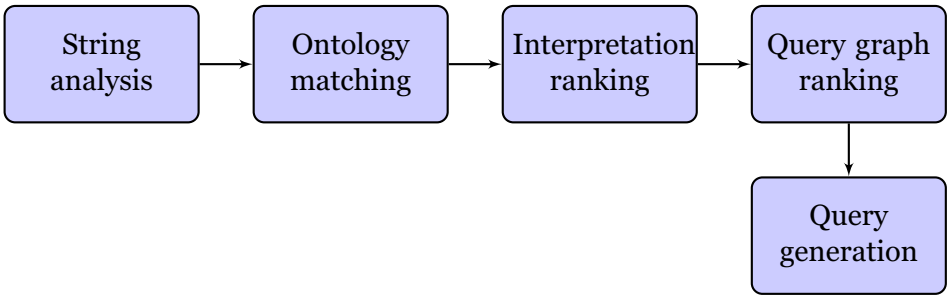



Figure 5.1: System architecture overview.

## 5.2 System architecture

When planning the implementation of OptiqueNLQF it was important to keep the different parts of the system separated, so if we wanted to do minor or major changes, this could be done without interfering with the other parts of the system. Therefore it is natural to approach the parts as individual working components, and this is the approach we have chosen when we wrote the code, and the writing of the thesis itself.

This system is not a substitute for the OptiqueVQS system, but rather an additional natural language component. This system communicates with the Optique system and takes advantage of some of the already implemented functionalities. In the future it is intended that the user will be able to use OptiqueNLQF and OptiqueVQS coherently.

### 5.2.1 Overview

The OptiqueNLQF-system consists of five main parts, as is shown in the system architecture overview in Fig. 5.1. Each part does its particular job, and passes its result on to the next part. *String analysis* controls how the sentence is parsed, and contains the main natural language processing components. *Ontology matching* is the part of the system that performs the searching. In this component the system searches the ontology for matching words and phrases. *Interpretation ranking* puts together and selects the concepts, relations and facets that represent the sentence in the best way with the lowest score, given the ontology matching output. *Query graph ranking* extracts one final combination of these objects. Lastly, after the correct information is retrieved, the *query generator* makes it into a SPARQL-query that can be executed on the database.

### 5.2.2 The Algorithm

OptiqueNLQF's main task is to receive a user's input information request, perform a query execution and present the user with the correct results. The initial input for the system is the user's information request which is passed to the string analysis where the input question is split into phrases. A term "phrase" will be used a lot and means a sequence of words: in our system it can be everything from a single word to the complete sentence. The string analysis part creates a tree data structure for representing the phrases, and sends the data structure to the ontology matching part. Here each phrase is individually matched against all the objects in our ontology. This matching is done by calculating the phrase's edit distance to the names of the ontology objects. Each phrase keeps a ranked list of its $N$-best ontology matched objects where the best match is the one with the lowest edit distance, 0 being a perfect match. Now there is a tree with nodes containing lists of ontology objects. The interpretation ranker traverses the tree, comparing each node's score to the combined score of this nodes children. If the children have a lower combined score, they take the current nodes place in the tree. When calculating this with a bottom up approach, we will eventually be left with a top node containing the $N$-best combinations of node objects, with lists of ontology objects, which together represent the initial user input sentence. These $N$-best objects will then be passed to the query graph ranker where the $N$ findings for each initial phrase again will be ranked using its relation to the other objects from the ontology. The connection distance in the ontology is what decides which objects is selected as the final input for the query generation. Only one of each ontology object from the previous combination of $N$ lists are passed on

from here, including, if there are any, ontology objects missing to create a connected graph based on the ontology concepts and relations. Finally these objects are selected, based on ontology type, and written as triples in the outputted SPARQL query with the proper prefixes and filters. The SPARQL query is then presented to the user in the OptiqueNLQF interface, before it is executed on the Optique system.

# Chapter 6

# String Analysis

## 6.1 Introduction

String analysis is the first and initial step in the OptiqueNLQF-system. This is where the user's input information request is analyzed. This step is of great importance and can in many ways severely decrease the systems complexity and time use. Since the ontology contains concepts, relations and attributes that may be a composition of several words, we have to perform searches on the substrings of the input string, from the full sentence and down to every individual word. However, a good natural language interpretation can reduce the number of phrases passed on by the string analysis component by omitting the redundant substrings of the user's input. Handling this problem was one of our starting points for the system.

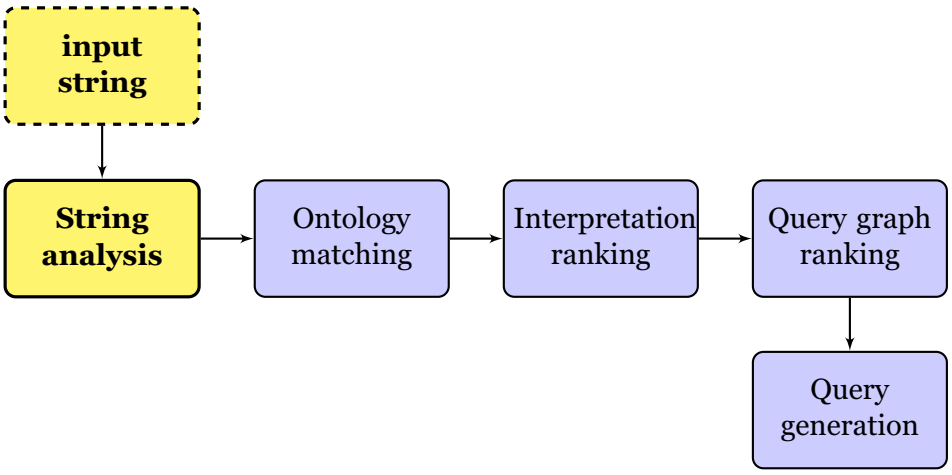



Figure 6.1: System architecture overview: String analysis.

## 6.2 Design choices

In this part of the system we have created two different approaches. It is clear that the string analysis is a place where changes can have great

impact on the final results of the query formulation. To have different implementation choices available to switch between provides us with the possibility to test the system with different run configurations, and we will get more data in the final testing phase, which is helpful for doing a more extensive evaluation of the system.

From the user's input string, handled by this part of the system, we are interested in locating the most relevant substrings, or phrases, before passing them on to the system's next part, the ontology matching. The most relevant substrings are the strings that are most similar to an object found in the ontology. From our example sentence "Fields operated by Statoil Petroleum AS", the optimal phrases would be: "Fields", "operated by" and "Statoil Petroleum AS". Knowing how to locate the correct phrases as fast as possible and avoiding the need to handle irrelevant phrases are the main difficulties of the string analysis part of the system. We have kept these goals in mind while deciding how to handle the string analysis part and during the implementation of what we have called the parse-tree method and substrings method.

### 6.2.1 Parse-Tree

This method uses a natural language toolkit to syntactically analyze the input information request, and generate a parse tree of the sentence. This provides us with a suitable data structure for the sentence, and is also a way to filter out the most relevant sub-phrases of the input string. By reducing the number of phrases extracted from the sentence, we will greatly reduce the number of actions done throughout the systems algorithm. We tested out several different toolkits, but the library we decided to go for was the Apache OpenNLP library [1]. This was chosen because Optique is a JAVA-project and OpenNLP is JAVA-based, whereas NLTK [2] for instance is Python-based [3]. Another factor was the legal reasons: OpenNLP is licensed [4] for commercial use, and since we are working with something that can be commercialized in the future, OpenNLP was the most sensible choice.

#### OpenNLP

OpenNLP is a JAVA-based natural language processing toolkit. For OptiqueNLQF we need to do a complete parse of the sentence for generating the parse-tree structure. When we analyze our example sentence "Fields operated by Statoil Petroleum AS" with OpenNLP it produces the tree shown in Fig. 6.2. In this tree every node represents a sub-phrase from the sentence, and they are given phrase tags or word tags, POS-tags accordingly. It is these sub-phrases of the sentence we are

[1] https://opennlp.apache.org/
[2] http://www.nltk.org/
[3] https://www.python.org/
[4] http://www.gnu.org/copyleft/gpl.html

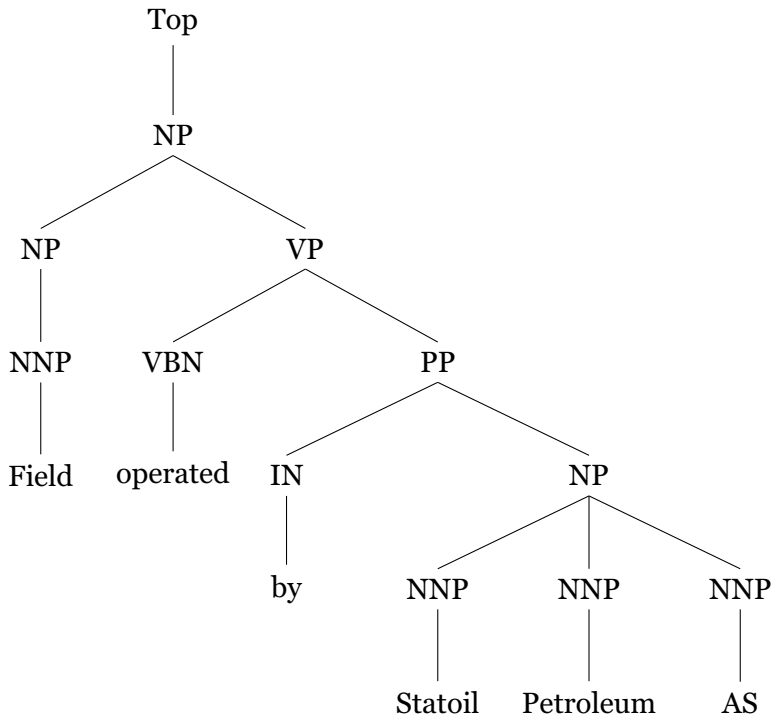



Figure 6.2: Example of parse-tree data structure.

interested in, and it is them we perform the string matching with against the ontology.

The sub-phrases from our example sentence are shown in Fig. 6.3. The phrase problem was encountered in the planning part when we looked at the last part of the example sentence “Statoil Petroleum AS”. We could immediately see that this part of the sentence would be a natural phrase, and we needed to find a smart way to identify what parts of the sentence would be best suited as phrases, and where a single word would be most correct. We wanted to use the parse-tree to locate noun-phrases, because nouns, especially names, are often represented by more than one word and is the most common occurrence of several words with natural coherence. However we realized that it would be sensible to apply this to the whole sentence, as it might also prove useful for relation phrases like “currentFieldOperator”, “taskForCompany” or “corePhotoForWellbore”.

A tree structure like this one will branch out from the top node, throughout the complete sentence, and down to the individual words. Let’s say we skip the occurrences where a phrase only has one child, which always will represent the same phrase as its parent. We know that a tree can have several children, but for the tree to generate the highest amount of sub-phrases every node must have no more than two children, it has to be a binary-tree [5]. By using these two rules we know that the highest possible

[5]http://en.wikipedia.org/wiki/Binary_tree

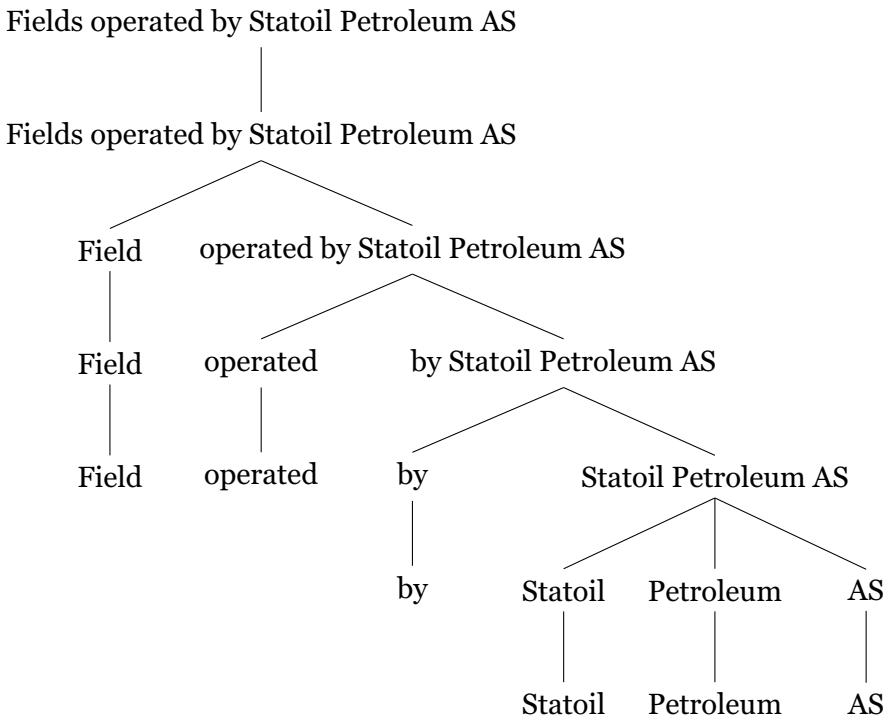



Figure 6.3: Example of parse-tree data structure.

count *N* of sub-phrases generated from a parse-tree (p) is:

$$N_p \leq 2n - 1$$

Using our example sentence containing 6 words(n) this results in *(2*6)-1 = 11*. But the example sentence is not binary: it has a phrase that generates three children, "Statoil Petroleum AS", therefore the total number of phrases, not counting unary representations, in our example sentence will be 10.

### Limitations of using the parse-tree method

There are some issues with the parse-tree method previously described. The OpenNLP toolkit requires a large 36 MB file containing the language model in order to parse and analyze the input sentence. This is a great deal and it takes the method just about 5 seconds to do a complete analysis of the sentence. Another problem is the lack of completeness regarding possible phrase pairs, and especially on occasions where a verb is followed by a preposition. This particular incident occurs in our example question where the words in "operated by Statoil Petroleum AS" do not form the phrases"operated by" and "Statoil Petroleum AS". Instead "Operated" is set to be a daughter of the former phrase, with "by Statoil Petroleum AS" as its sibling. One possible solution to this problem could be to force the language parser to treat all occurrences of verb + particle and verb + preposition as what is called phrasal verbs, described in Foundations

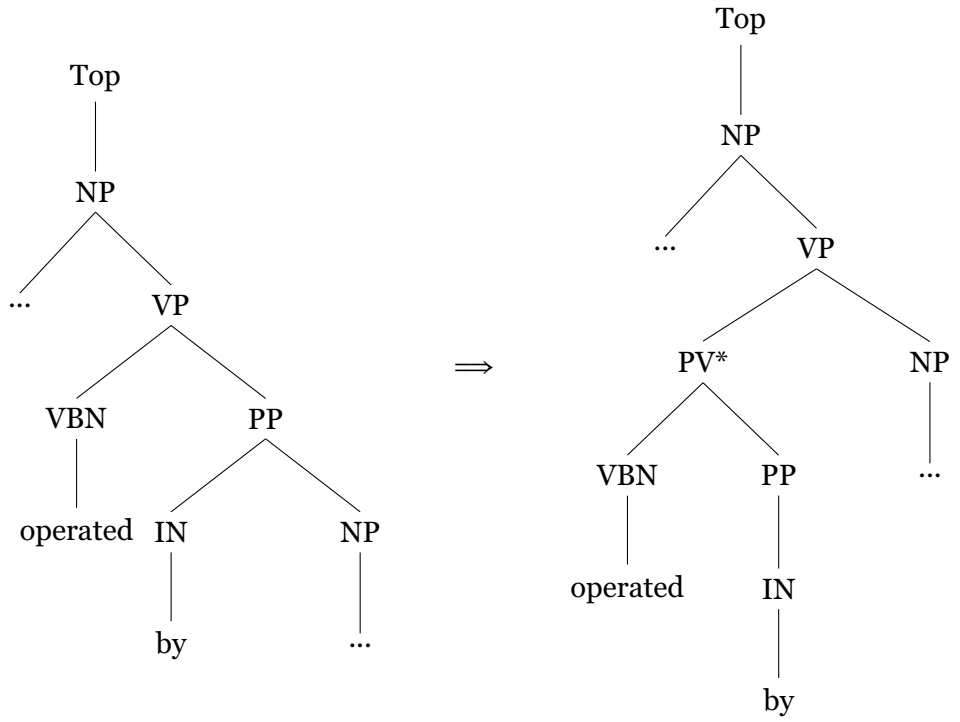



Figure 6.4: Phrasal verb parse proposition.
"..." indicates parts of the tree that is similar in both trees.
The tree to the left is similar to the Fig. 6.2.

of Statistical Natural Language Processing [12, pp. 96–97]. OpenNLP does not identify phrasal verb occurrences as a unit, instead treating them like verb + possible remaining sentence. By forcing this phrasal verb structure upon sentences containing verbs + preposition / particles the tree structures would change and be more suited for ontology matching. An example of this is shown in Fig. 6.4, where the tree on the left illustrates how a string is handled by the OpenNLP parser, and the tree on the right illustrates how a string might have been handled using a phrasal verb identifier. Note that the prepositional phrase (PP) node has switched to be a child of the new phrasal verb (PV) node, and now the words "operated by" forms the center phrase of the sentence. Though an interesting idea with potential to manage the phrasal verb problem, to achieve this would require that we change the OpenNLP parser structure, and we simply did not have the time to implement our theory. We did however experiment with a node-shifting JAVA-method to perform these actions in the OptiqueNLQF, but this proved rather unsuccessful.

### 6.2.2 Substrings

The parse-tree method generates most of the needed phrases. It also provides additional potentially useful information about the words, phrases and the sentence. However, we wanted to create another implementation of the string analysis problem. Since the current version of the parse-tree method failed to provide us with all the phrases we needed, we wanted to find all substrings from the sentence, and perform a search on them all. The time-use of reading the language model is currently so high that the effect of doubling the number of phrases will actually decrease the run speed of the system, compared to parse-tree method.

To extract all sub-phrases and still get a sensible data structure for the ontology matching we decided to make a string pyramid building up from the individual words, adding onto each other generating all the phrases in the sentence. The OpenNLP toolkit is used to make a list of words from the user's input string using the tokenizer. The pyramid is structured as is shown in Fig. 6.5. The bottom row is a list of all the individual words in the input question, and has the size $n$ where n is the number of words. The second row is a list of tuples of the words in the sentence, and will have the size *n-1*. The third line in the pyramid will be a line of the triples of the words in the input sentence, and have the size *n-2*, and so on until the top where the list contains one element, which is the complete sentence and has the size *n-(n-1)*, which is 1. To calculate how many words are in a given row $m$, with a given sentence length $n$ we have the formula *n-(m-1)*. We can use this further by stating that the sum of all rows is given by:

$$(n-(1-1))+(n-(2-1))+\ldots+(n-(n-1))$$

$$\Downarrow$$

$$\sum_{k=1}^{n}(n-(k-1))$$

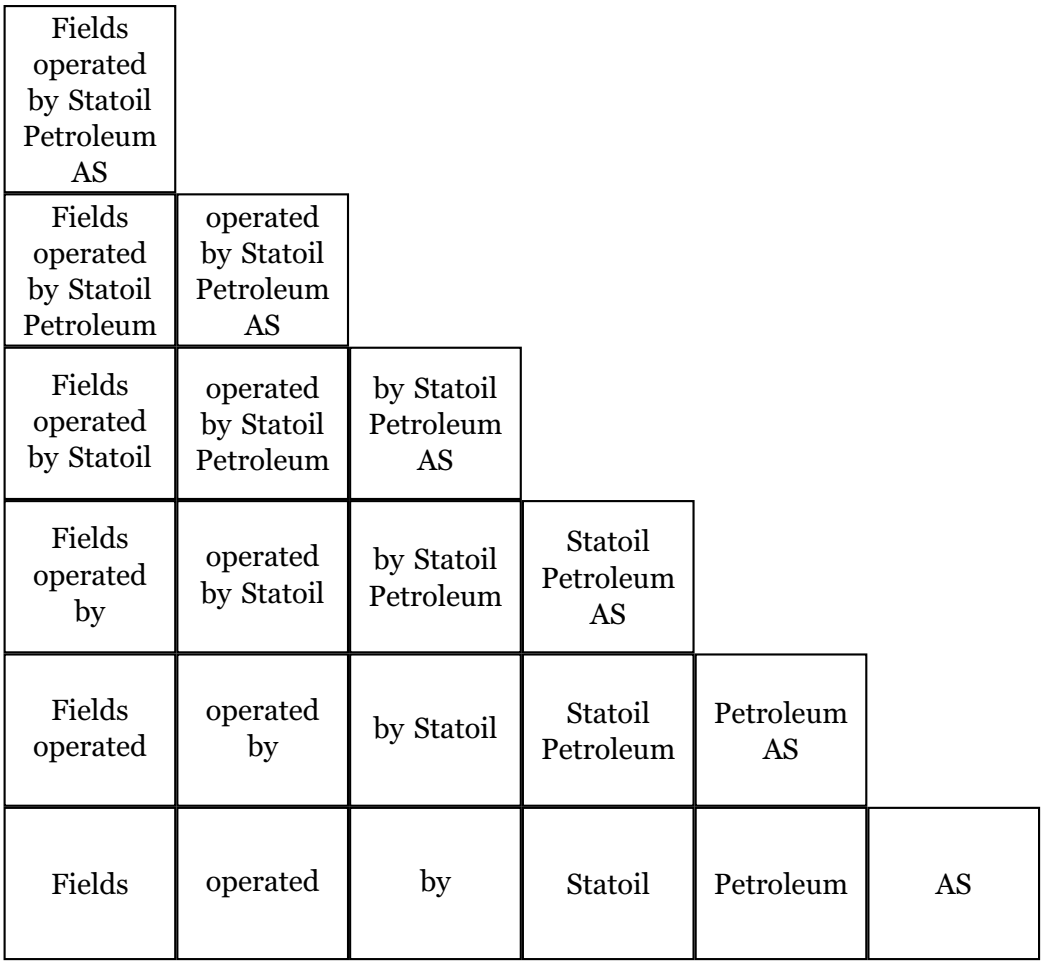



Figure 6.5: Substring pyramid.

Now it is the matter of finding the sum of a finite sequence of numbers. For this we can use the Gaussian sum [2], and we can write our formula for getting the maximum number ($N$) of sub phrases using the substrings (s) method on the form:

$$N_s \leq \frac{n(n+1)}{2}$$

Going back to our example sentence which has 6 words, we now know that the total number of substrings extracted from this string will be:

$$\frac{6 * (6+1)}{2} = 21$$

So, using the substrings method, we end up with 21 sub-phrases from our example sentence. Whereas with the previously explained parse-tree method we had 18 sub-phrases, but since a lot of these are duplicates, and we simply can skip the phrase when we know it is similar to the preceding one, there were 10 unique phrases given from the parse-tree method. This number will vary on word type, structure and length, but it will be considerably lower than the number of sub-phrases we get if we want to use the substrings method. We also know that the total number of phrases will increase faster using the substrings method than the parse-tree method, so the difference will increase when sentence length is increased. Even though the sentences are short in the prototype stage, we must think forward and prepare the system for longer, more complex sentences.

Getting all substrings from a string is a very brute force way of solving a problem, but it is nevertheless an interesting baseline for testing other ways of selecting which phrases to use.

## 6.3 Main differences

The two main measurements for the task of finding the correct sub-phrases of the sentence is speed, the time it takes the system to run, and recall [6], how many of the correct phrases was retrieved by the system. Concerning recall, the substrings method will always get a 100% recall score, since it retrieves all possible sub-phrases and therefore always will find every correct phrase. This method will, however get a very low precision score. Assuming that “Fields”, “operated by” and “Statoil Petroleum AS” are the correct phrases from our example sentence, we have three correct phrases. The substrings-method will get a precision and a recall score of:

$$Precision = \frac{relevant\ phrases \cap retrieved\ phrases}{retrieved\ phrases}$$

$$Recall = \frac{relevant\ phrases \cap retrieved\ phrases}{relevant\ phrases}$$

$$Precision = \frac{3}{21} = 14\%.$$

[6] http://en.wikipedia.org/wiki/Precision_and_recall

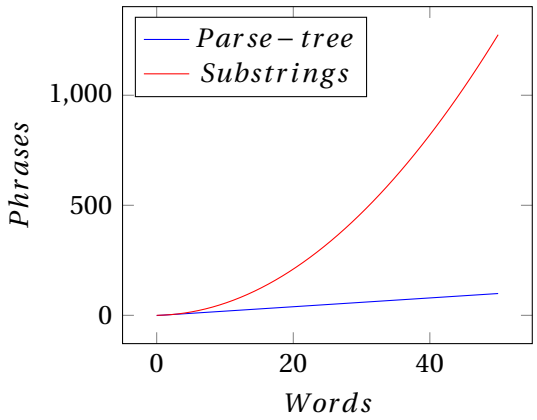



Figure 6.6: Phrases generated with different methods.

$$Recall = \frac{3}{3} = 100\%.$$

The parse-tree method will, using the same example sentence, get a precision and recall score of:

$$Precision = \frac{2}{10} = 20\%.$$

$$Recall = \frac{2}{3} = 66.6\%.$$

It is problematic that the parse-tree does not find all correct phrases. Implementing the phrasal verb theory will hopefully increase both recall and precision of the pare-tree method.

How fast the two methods will run is harder to evaluate. Precision will play a part here, since the number of phrases retrieved is highly relevant. As seen in Fig. 6.3 the quadratic growth of the substrings method is considerably higher than the linear growth of the parse-tree method. We have to assume that there is a number of operations that will have to be done on every phrase, so keeping this number down can prove to be important. We will come back to this issue in the following chapters. The constant 5 second parse-time for the language model gives the substrings method a head start, but let it be noted that this is a constant number, it has no noticeable rise according to sentence length, or number of sentences.

## 6.4 Possible Changes / Further work

For future versions of the system we know that applying a natural language understanding module can provide information on what word types a given word is. It would be prudent to add specifications to handle different POS-tags or apply word entity recognition so we can get an understanding on what a single word is supposed to describe given some pre-defined categories. This is one of the main reasons for sticking to the parse-tree method. Although the substrings method is the fastest one, provided we

use smaller 5 - 10 word sentences, the parse-tree implementation has its advantages and also provide us with many possibilities when the system is expanded and a more complex sentence evaluation is needed. Adding the opportunity to include conjunctions to the question, and then in particular “and” and “or” is certainly one of the main focus points for further work. A parse-tree will in these cases naturally parse a conjunction as a link between two verb phrases or noun phrases, and split the sentence into phrases with the conjunction as the divider. The conjunction and the phrases connected by the conjunction will carry with them this information, and in the final query generation-part the proper steps will be taken to write the “and” or the “or” into the query. Negations in a sentence will work in a similar way. When an adverb is recognized as a negation, a variable is passed with the strings adding a negation filter to the final query. We are also interested in possibly retrieving multiple parse tree representations from the parse-tree method. This is not currently possible using the OpenNLP system, but it would provide us with an optional phrase composition that could be used if the first one should prove not to contain the optimal interpretation of the sentence.

# Chapter 7

# Ontology Matching

## 7.1 Introduction

The ontology matching is the second part of the OptiqueNLQF-system and it is the main search component. This is where the system communicates with an ontology, uses the phrases generated in the string analysis part and retrieves data. Performing this task is mostly string matching, using what we know and trying to find the most similar data from the data set we have. This is also the point of the system where we step away from strings and phrases and start handling objects representing data from the ontology. We are searching for the objects whose name is most similar to the phrases given from the previous part of the system.

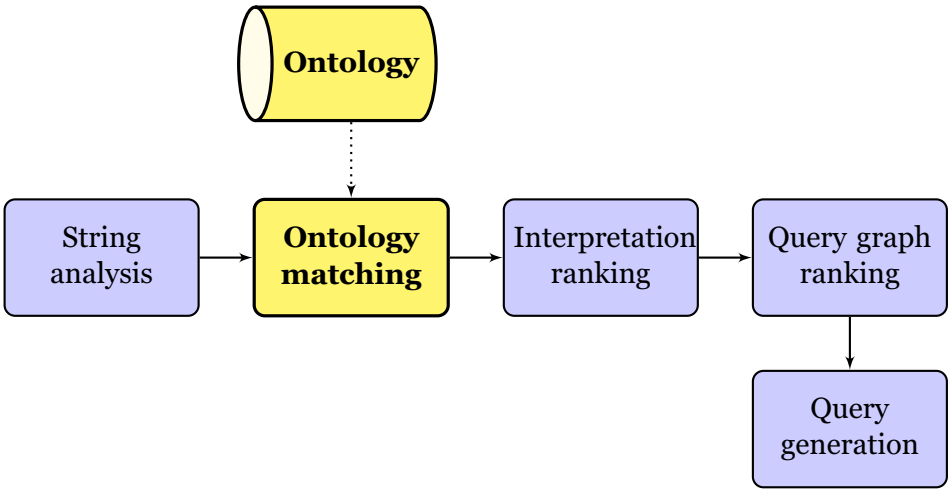



Figure 7.1: System architecture overview: Ontology matching.

### 7.1.1 Ontology

The system parses an ontology which is a specification of the concepts describing information in the database, the relations between the concepts and the attributes/facets [4] associated to the individual concepts. Concepts represent specific things like “Field”, “Company” or “JacketTripod-

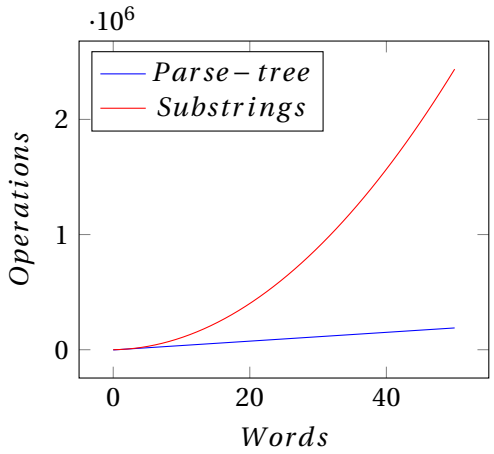



Figure 7.2: Number of ontology-matches done by the substring and Parse-tree methods.

Facility". Attributes/facets are datatype properties with possible ranges or values. The relations are relationships between the concepts, object properties in OWL [1]. The loading and parsing of the ontology is performed by a previously existing method in the Optique system, so OptiqueNLQF is making use of this method. We use an already integrated part of the OptiqueVQS [2] system, and it is important that OptiqueNLQF use the same methods as the OptiqueVQS and that the systems are compatible for further integration in the future. When the ontology is loaded it is parsed and we retrieve the information, which is returned as JSON-objects [18]. The ontology which is being used in this project is an annotated test-ontology based on Oljedirekoratet's NPD [3] factpages [16]. This ontology contains 213 concepts, 736 relations and 962 attributes. Together these groups sum up to 1911 searchable objects from the ontology.

## 7.2 Implementation

The ontology matching in OptiqueNLQF is called from the preceding string analysis part every time a phrase is located. A phrase is passed to the ontology matching class, and it returns a list of the three objects from the ontology that is most similar to the phrase. With there being so many objects in the ontology, and many of them having similar names, or parts of the name being identical, we have to return the $N$-best findings. The fact that we use three is just a starting point, and further testing to find out if there should be more or less returned matches is recommended. Changing this will, however, be an uncomplicated procedure.

All the phrases from the string analysis is compared with all the objects from the ontology. All return its 3-best list, no matter how good or bad

[1] http://www.w3.org/2001/sw/wiki/OWL
[2] http://fact-pages.fluidops.net/resource/VisualQueryFormulation
[3] http://factpages.npd.no/factpages/Default.aspx?culture=no

they are scored by the system. We do not yet know which set of phrases form the lowest scoring combination of ontology objects, and even though one phrase gets a high score it can still be a part of the lowest scoring set. Fig. 7.2 shows the difference in operations done by the ontology matching part based on which method was used in the string analysis part. Given our example input sentence “Fields operated by Statoil Petroleum AS” and the phrases generated by the two different implementations in the string analysis step we can calculate how many operations the ontology matching have to perform. From the parse-tree method there was 10 distinct phrases passed on, which results in:

$$10 * 1911 = 19110 \textit{ operations.}$$

This is easy for a computer to handle, and it is done quickly. Using the substring method for the string analysis task the number of phrases is more than doubled, and is now 21, which results in:

$$21 * 1911 = 40131 \textit{ operations.}$$

This is also handled very quickly by a computer, but as we mentioned in the last chapter, the number of phrases passed on by the substring method has a noticeably higher growth than the parse-tree method, especially when the sentence gets longer. In the introduction chapter we presented a user example question Ref. 1.3, retrieved through the Optique user testing, which contains two sentences, each containing respectively 33 and 18 words. That is if we presume that the system separate the input strings when getting to a period. (The parse-tree method does this automatically and it is easy implemented into the substring method.) The parse-tree method will in this case generate a maximum of:

$$((2 * 18) - 1) + ((2 * 33) - 1) = 35 + 65 = 100 \textit{ phrases}$$

$$\Downarrow$$

$$100 * 1911 = 191100 \textit{ operations.}$$

This is 10 times the number of operations compared to the other example, this is not a dramatic increase in operations considering the word count increased from 6 to 41. The substring method on the other hand will generate:

$$(\frac{(18 * (18 + 1))}{2}) + (\frac{(33 * (33 + 1))}{2}) = 171 + 561 = 732 \textit{ phrases}$$

$$\Downarrow$$

$$732 * 1911 = 1398852 \textit{ operations.}$$

The difference in numbers of operations is evident, the substring method has to do more than 7 times the operations of the parse-tree method. The substring method also has to do 35 times the number of operations compared to the previous example, it increases much more than the parse-tree method. This example is however, for now, of the unlikely kind and handling these words would probably cause more harm than good at this stage.

### 7.2.1 String matching

Calculating the similarity of a phrase and of a name of an ontology object is done using the edit distance [4] of the two strings, or more precisely, calculating their Levenshtein distance [11, p. 53–55]. This is a method of comparing two strings using three operations i) inserting a character ii) removing a character, and iii) replacing a character by another character. Each of these operations has a cost of 1, and the Levenshtein distance between two strings is the total number of operations that have to be done on the strings before they are equal. When our phrase "Fields" from the example sentence is compared with the concept name "Field", it will get the Levenshtein distance of 1. The top three lowest scoring ontology objects for "Fields" is:

1. "Field" - concept - 1.0
2. "Well" - concept - 4.0
3. "FieldArea" - concept - 4.0

"Fields" has the score 1 since the only operation needed is to remove the "s" in the end of the input phrase "Fields". "Well" gets a score of 4.0 since the first letter "F" is removed, the second letter "i" is replaced with a "W" and the final letter "d" is replaced with an "l". "FieldArea" also has en Levenshtein distance of 4.0, the final "s" in "Fields" is replaced with an "A", and the three last letter is added, to get "FieldArea".

In the example above with the three matches for "Fields" the only object type that was returned was concepts, and the sentence "Fields operated by Statoil Petroleum AS" does not contain all ontology object types. We will therefor look at another example sentence: "Current field operators of Ekofisk". The produced sub-phrases and scores from this sentence can be seen in Tab. 7.1. As we can see from the table each phrase, in bold, is followed by its three most similar objects from the ontology. In column one, below the phrase itself, is the string that the given phrase was matched with. Note that the objects of the type attribute do not have their matched string in column one, but what type of attribute was found, for example *name*. In column two is the Levenshtein distance between the phrase and its matched string. Only attributes and relations have something in column three, here they have what concept the ontology object is related to. Only relations have something in column four, this is the concept that is related to the concept that is listed in column three. In column five is the object type of each finding: concept, relation or attribute. Some of the relations is marked with a ^ at the beginning of its name, this represents that it is an inverse relation. "operator" is the only phrase that has a concept, relation and an attribute in its top three list, but it had a string similarity of 4.0 with both the relation "wellOperator" and an attribute "facilityType", and it is likely that there are several other objects that has a string similarity of 4.0 with "operator". It is interesting to observe the phrases "field operator",

[4] http://en.wikipedia.org/wiki/Edit_distance

| | | | | |
|---|---|---|---|---|
| **Current** | | | | |
| Agent | 4.0 | | | Concept |
| Survey | 4.0 | | | Concept |
| Quadrant | 4.0 | | | Concept |
| **Current field** | | | | |
| currentFieldOwner | 6.0 | Field | ProductionLicence | Relation |
| ^currentFieldOwner | 7.0 | ProductionLicence | Field | Relation |
| wellboreType | 8.0 | InitialWellbore | | Attribute |
| **Current field operator** | | | | |
| currentFieldOperator | 2.0 | Field | Company | Relation |
| ^currentFieldOperator | 3.0 | Agent | Field | Relation |
| ^currentFieldOperator | 3.0 | Company | Field | Relation |
| **Current field operator of** | | | | |
| currentFieldOperator | 5.0 | Field | Company | Relation |
| ^currentFieldOperator | 6.0 | Agent | Field | Relation |
| ^currentFieldOperator | 6.0 | Company | Field | Relation |
| **Current field operator of Ekofisk** | | | | |
| currentFieldOperator | 13.0 | Field | Company | Relation |
| ^currentFieldOperator | 14.0 | Agent | Field | Relation |
| ^currentFieldOperator | 14.0 | Company | Field | Relation |
| **field** | | | | |
| Field | 0.0 | | | Concept |
| Well | 3.0 | | | Concept |
| Oil | 3.0 | | | Concept |
| **field operator** | | | | |
| fieldOperator | 1.0 | Agent | Company | Relation |
| FieldOperator | 1.0 | | | Concept |
| fieldOperator | 1.0 | FieldOperator | Company | Relation |
| **field operator of** | | | | |
| fieldOperator | 4.0 | Agent | Company | Relation |
| FieldOperator | 4.0 | | | Concept |
| fieldOperator | 4.0 | FieldOperator | Company | Relation |
| **field operator of Ekofisk** | | | | |
| fieldOperator | 12.0 | Agent | Company | Relation |
| FieldOperator | 12.0 | | | Concept |
| fieldOperator | 12.0 | FieldOperator | Company | Relation |
| **operator** | | | | |
| Operator | 0.0 | | | Concept |
| wellOperator | 4.0 | DevelopmentWellbore | Company | Relation |
| facilityType | 4.0 | FixedFacility | | Attribute |
| **operator of** | | | | |
| Operator | 3.0 | | | Concept |
| operatorForField | 6.0 | FieldOperator | Field | Relation |
| operatorForField | 6.0 | Operator | Field | Relation |
| **operator of Ekofisk** | | | | |
| operatorForField | 9.0 | Agent | Field | Relation |
| operatorForLicence | 9.0 | Agent | ProductionLicence | Relation |
| operatorForField | 9.0 | FieldOperator | Field | Relation |
| **of** | | | | |
| reservesResourceClass | 1.0 | DiscoveryReserve | | Attribute |
| reservesResourceClass | 1.0 | DiscoveryReserve | | Attribute |
| mudType | 1.0 | WellboreDrillingMudSample | | Attribute |
| **of Ekofisk** | | | | |
| name | 5.0 | Pipeline | | Attribute |
| name | 5.0 | OilGasPipeline | | Attribute |
| name | 5.0 | CondensatePipeline | | Attribute |
| **Ekofisk** | | | | |
| name | 0.0 | FieldReserve | | Attribute |
| name | 0.0 | Field | | Attribute |
| name | 0.0 | Reserve | | Attribute |

Table 7.1: Results of string matching on “Current field operators of Ekofisk”.

"current" and "Ekofisk". All three have an equal string similarity for all their top three matches, and again there are most likely other objects that might just as well be in the list. This also illuminate another problem: which one is the correct one? As for "Ekofisk", it is a name for both a "field", "FieldReserve" and a "Reserve". "Current" has Levenshtein distance of 4.0 with "Agent", "Survey" and "Quadrant", and these concepts represent very different things. Resolving this problem will be discussed in the next two chapters, the interpretation ranking chapter and the query graph ranking chapter.

## 7.3 Other functionality

The edit distance calculation can be changed in several ways to improve the matching. The method can handle the input string as it is, it will then separate between capital and lower-case letters, even if they are the same letter like "f" and "F". Or the method can make all strings into lower case letters (or upper case, as one prefers). This can possibly make a difference in what objects are chosen for each phrase. All concepts in the ontology begins with a capital letter and all relations begin with lower-case letters. Attributes are a little of both, where names always start with a capital letter, but types and data types often begin with lower-case letters. This can cause difficulties for instance since starting a sentence with a capital letter must still be assumed to be common amongst future users. For our project we have implemented an ignorecase() option, an easy way to switch between these two functionalities, and this will be tested and evaluated on the test-set questions. Let it be noted that the findings in Tab. 7.1 are retrieved with the ignorcase() activated. Another way to possibly improve the edit distance is to make it sensible to the camel cased [5] relation names. "current field operator" gets an edit distance of 2.0 to the relation "currentFieldOperator" in Tab. 7.1, and it would even get a score of 4.0 if the ignorecase() was inactive. This is problematic since it obviously is the correct ontology object, and it should get the lowest score possible. One way of handling this is to recognize a capital letter in the middle of the name of the ontology object. When this occurs we make the letter lower case and add a space before calculating the edit distance. A function for handling such an operation is not implemented in the system since it was not considered an immediate issue. It is however important to keep these matters in mind if the system is to be improved further.

### 7.3.1 Levenshtein distance variation

The Levenshtein distance's third operation option, iii) replacing a character by another character, may in some cases cause another object than the one intended to be retrieved. This because the replace action can be seen as the same as the two other Levenshtein rules combined: remove one character and insert another. One way of dealing with this problem is if the replace

[5] http://en.wikipedia.org/wiki/CamelCase

| Levenshtein distance with replace | | |
|---|---|---|
| 1. | “field” - “well” | 3.0 |
| 2. | “field” - “fieldArea” | 4.0 |
| Levenshtein distance without replace | | |
| 1. | “field” - “fieldArea” | 4.0 |
| 2. | “field” - “well” | 5.0 |

Table 7.2: Results for Levenshtein distance with and without replace.

operation will have a scoring penalty of 2, instead of 1, or just remove the possibility of using the replace rule completely. If we look at the word “field” and it’s matching results with the words “well” and “fieldArea” we will see the difference in the two methods, shown in Tab. 7.2. Using the regular Levenshtein distance method “well” will get a score of 3.0, after the operations remove “f”, replace “i” with “w” and replace “d” with an “i”. The word “fieldArea” will get a score of 4.0, after adding the last four letter to the original word. Using the alternative Levenshtein distance method, the same words would be in opposite order. “FieldArea” will still score 4.0 after adding the four letter, but “well” will now score 5.0. Now the method has to remove the “f” and the “i”, and add the “w”, remove the “d” and add the final “l”. In this example the two concepts switch places, and this can have great impact on the final generated query.

### 7.3.2 Stopwords

One thing that can cause problems in the upcoming stages of the system is occurrences of words that don’t have any special meaning for the sentence, and don’t provide any information on its own. These words are called “stopwords” [7, p. 806], and there are two main ways of handling them; keep them, or remove them. If we look at Table. 7.1 the phrase “of” has a string similarity 1.0 with three attributes, and most likely many more. Given the sentence “Current field operators of Ekofisk” the “reservesResourceClass” of a “DiscoveryReserve”, or the “mudType” of a “WellboreDrillingMudSample” will most likely be completely irrelevant for the query’s meaning and for the information the user needs to retrieve. By removing the stopwords we are removing possibly irrelevant phrases, thereby decreasing the number of operations needed to perform. If the stopwords are useful it is most likely in combination with other words, by creating a phrase and matching it with a relation in the ontology. We have therefor implemented a *removeStopwords*-method which can be activated after the phrases are identified, to enable the opportunity to perform test runs on the system both with and without stopwords.

## 7.4 Further work

There are many ways of improving this part of the system. The Levenshtein method is one way of handling string similarity, and we have already gone

over some aspects of the method that can be changed. The camel case action is a step towards better matches for phrases similar to relation names. There are many other ways of handling the string matching. There are alternatives to the Levenshtein distance, for instance a string matching automata [8] [6]. The automata will use the letters in the ontology object names as the automata's alphabet and run the phrases located in the string analysis part as the finite input string. To implement another method for string analysis would be interesting, as we are always searching for ways to make improvements.

[6] http://en.wikipedia.org/wiki/Automata_theory

# Chapter 8

# Interpretation ranking

## 8.1 Introduction

Interpretation ranking is the third main part of OptiqueNLQF. After the phrases are extracted from the user's input sentence and these phrases are matched with objects from the ontology, a ranking algorithm is executed to get the lowest scored ontology objects. The goal for this algorithm is to find the combination of ontology objects which together form the initial input sentence with the lowest combined score. This is where the filtering of objects begins and removing redundant objects that might have been selected at an earlier stage, and we are now at the start of locating the objects that together will form our final query.

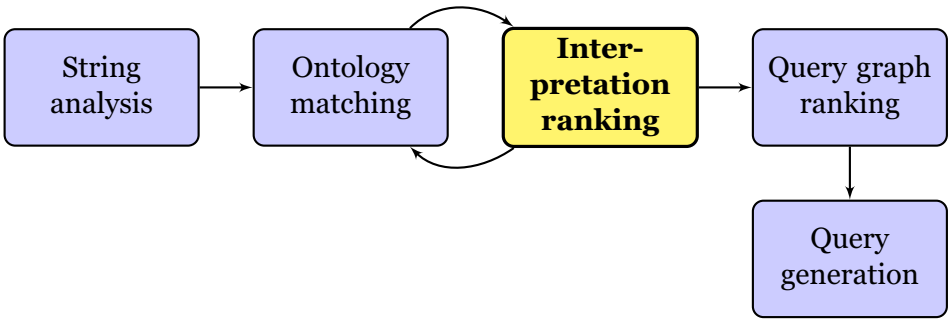



Figure 8.1: System architecture overview: Interpretation ranking

## 8.2 Implementation

This part of the system is highly based on the string analysis part, and the way we chose to represent the phrases in the previous parts will be put to good use at this stage. Since we have two different ways of handling the string analysis, there are also two ways of handling the interpretation ranking of the phrases. The ranking methods are however quite similar, and are built using the same general idea: matching one node to its children and passing on the one with the lowest combined score. The methods also return the same type of data structure in the end, making it easy to switch

from one to the other. Also, notice that in isolation both methods in use produce the same result given that the similar, lowest scored phrases are retrieved for the two methods.

### 8.2.1 Tree method

**Algorithm 1** Recursive tree algorithm

```
1:  procedure TREERANK(Parse)
2:      ArrayList values ← query.getValues(Parse)
3:      valueMap.put(Parse, values)
4:      ArrayList bestList
5:      childrenScore ← 0
6:      if Parse.getChildCount() == 0 then
7:          ArrayList temp ← valueMap.get(Parse)
8:          return new Combined(valueMap.get(Parse).getScore(), temp)
9:      for Parse child : Parse.getChildren do
10:         Combined temp ← TREERANK(child)
11:         childrenScore+ = temp.getValue()
12:         bestList ← temp.getObjects()
13:     if childrenScore ≤ valueMap.get(Parse) then
14:         return new Combined(childrenScore, bestList)
15:     else
16:         ArrayList temp ← valueMap.get(Parse)
17:         return new Combined(valueMap.get(Parse).getScore(), temp)
```

When the system runs the parse-tree method the objects are ranked by a recursive method that calls on the children of each node, starting at the head of the parse tree. The algorithm implemented in OptiqueNLQF is shown in pseudocode in Alg. 1. The argument of the treeRank() method is called Parse. In this context Parse comes from the OpenNLP library and is a data structure containing information about a sentence that has been parsed using the OpenNLP toolkit. This data structure contains several built-in methods which have been implemented in the system. The treeRank algorithm uses two: getChildCount() returns the number of children connected to a node; while getChildren() returns a list of the nodes' children. *Combined* is a data structure we have created for keeping track of the score of the objects, as well as a list of the objects. This is also the data structure used for keeping track of the selected objects from here on out in the system. There are also some other methods we have created that is seen in the pseudocode, getValue() and getObjects(), the function of which is respectively retrieving the scores and objects connected to the Parse. As the method runs it uses the data stored in the valueMap which can also be seen in the pseudocode. We find the call to the string matching part of the system (query.getValues()) described in the previous chapter in the first two lines of the pseudocode, and this is why an arrow pointing back to the ontology matching is added to the system architecture overview in Fig. 8.1.

| | | | | |
|---|---|---|---|---|
| Current=4 | field=0 | operator=0 | of=1 | Ekofisk=1 |

Figure 8.2: The bottom line of the pyramid.

The algorithm for the treeRank() method works by starting with an if-test to see if the current node is one of the leaf nodes by checking if its child count is nil, Alg. 1[line 6]. If it is one of the leaf nodes the method returns a *Combined* object, Alg. 1[line 8], with the score and ontology object of the current node, to be added with the other siblings of the current node, if there are any. If the node is not a leaf node the algorithm continues to the for-loop and recursively calls the method, treeRank(), Alg. 1[line 10], on the current node's children. After the node have retrieved the scores and objects of its children it returns itself or the combination of its children to the parent node of the current node. When there is no parent node to return to the algorithm is complete. Then the top node, the one we started with, returns the combination, the *Combined* object, of the set of phrases which have the lowest combined score. The algorithm works its way down to the leaf nodes of the tree, and when it reaches the end, it goes back up again, combining the children of each node and continuously selects the nodes with the lowest scores.

### 8.2.2 Pyramid

The substring method from the string analysis part is built like a pyramid, and its phrases are ranked using the idea of the pyramid's composition. The idea behind the pyramid algorithm is that we start by looking at the scores of the bottom of the pyramid. These are the phrases containing every individual word, except now it is not the phrase, but a list of the *N*-best objects which matched the phrase, and the edit distance of these objects. From the sentence "Current field operator of Ekofisk" the bottom line will contain what is shown in Fig. 8.2. Note that this is a simplification of what it would look like, we only show the best result from each phrase and the objects are represented by their original phrase, not the retrieved ontology object. Calculating the scores for the first line is trivial, it can only contain the score of which the single word by itself is assigned.

The second line of the pyramid is calculated by checking the phrases located in the second line of the pyramid, all substrings containing two of the words from the sentence: "Current field", "field operator", ... ,"of Ekofisk", and the combination of this phrase's children. The children of the second line nodes are the node below and the node below and to the right. "Current field" will have "Current" and "field" as its children, and it is the combination of these two that will be compared to "Current field". This is represented in Fig. 8.3, each second line node will be compared with the combinations of the nodes whose arrows are directed at it. In our example, none of the two-word phrase scores are lower than any of

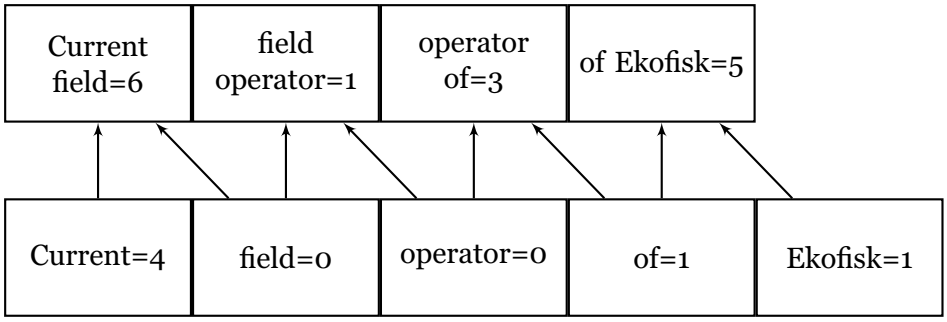



Figure 8.3: Calculation the second line scores of the pyramid.

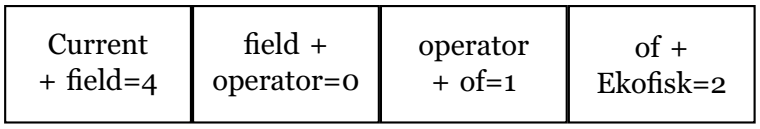



Figure 8.4: The second line of the pyramid.

the combinations of one-word phrases and therefore the second line will contain the combination of the first line's one-word phrases. Fig. 8.4 shows how the second line are represented.

The third line is slightly more complicated, because we do not combine the two nodes below the node we are currently in, but all combinations of the nodes building up to the sub-pyramid that has the third line node as its head. This means that when we are in the third row of the pyramid, we combine the node in the first row with the node below and to the right in the second row, and the node below in row two with the number three node to the right in the first row. As we can see from Fig. 8.5, this gives us three options for every third line node, which in this example would yield:

1. (Current + field) + operator = 4 + 0 = 4
2. Current + (field + operator) = 4 + 0 = 4

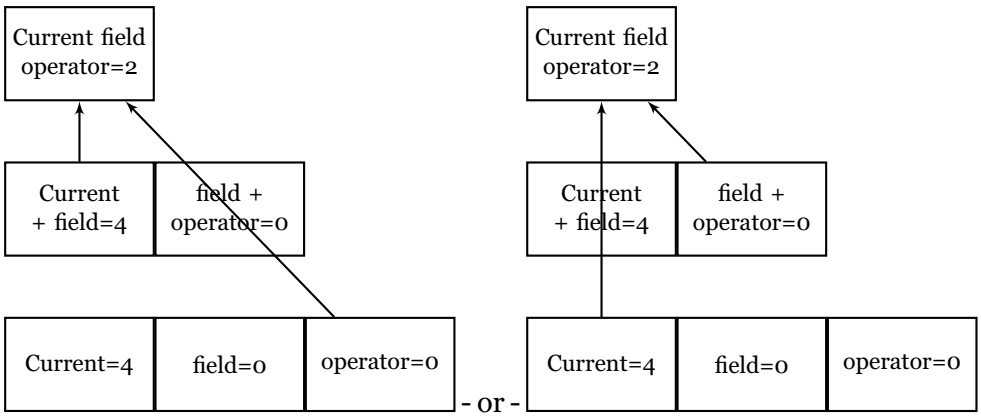



Figure 8.5: Calculation of line three in the pyramid.

3. Current field operator = 2

**Algorithm 2** Pyramid algorithm

```
1:  procedure PYRAMID(tokens)
2:      i ← tokens.length − 1
3:      j ← 0
4:      for i ≥ 0 do
5:          for j ≤ tokens.length do
6:              if pyramid[i][j] ≠ null then
7:                  current ← pyramid[i][j]
8:                  k ← i
9:                  for k ≤ j do
10:                     right ← pyramid[i][k]
11:                     left ← pyramid[k + 1][j]
12:                     if right + left < current then
13:                         pyramid[i][j] ← right + left
14:                     k ← k + 1
15:             j ← j + 1
16:         i ← i − 1
```

In this isolated sub-pyramid it's the third row node "Current field operator" that has the lowest score. From here on and throughout the pyramid the nodes are calculated by matching the possible combinations of the sub-pyramid the given node is the head of, following the same pattern as described. This means that the first fourth line nodes will have to check the combinations: first row first node and third row second node, second row first node and second row third node, third row first node and first row forth node, and finally itself. The pseudo code of how this is implemented in the system is presented in Alg. 2. When the loop is completed, the head node of the pyramid, containing the final objects, is returned and used as the input in the final part of the system.

Using the pyramid as the data structure is a sensible way of representing data that builds something together from smaller to bigger pieces, like in our case words that form phrases and eventually form a sentence. Our pyramid method and the algorithm for calculating the pyramid is greatly inspired by the CYK-algorithm [1] [7, p. 469–477]. This is a widely used parsing algorithm for context-free grammars, and it is a bottom-up parsing and dynamic programming algorithm. The CYK-algorithm is also very effective, and so is our pyramid algorithm. The pyramid algorithm has a complexity of $O(n^3)$, where n is the number of words in the sentence, this is manageable given our short sentences and is handled easily by the computer.

[1] http://en.wikipedia.org/wiki/CYK_algorithm

## 8.3 Possible changes / further work

In this part of the system what needs to be done is fairly straightforward and the actions are very dependent on what is done in the two previous parts. The data structure is based on past choices, and the biggest issue has been to decide how to design the algorithm concerning how the data should be properly traversed. Regarding the conceptual algorithmic choices there are, compared to the other system parts, not many pressing changes that need to be dealt with. Both the methods in use are fast and equally correct and do the job they are supposed to do. There are of course, as always, things that can be improved and the interpretation ranking part of the system is far from being complete.

# Chapter 9

# Query graph ranking

## 9.1 Introduction

The fourth part of OptiqueNLQF is the query graph ranking. This is where the final selection of objects take place, and this part communicates its returned values to the final part of this system, the query generator. In the previous system part we ranked the ontology objects by comparing their edit distance and selected the $N$-best object combinations for the sentence. Now we are interested in the objects' relation to each other in the ontology. We also further want to separate the irrelevant objects.

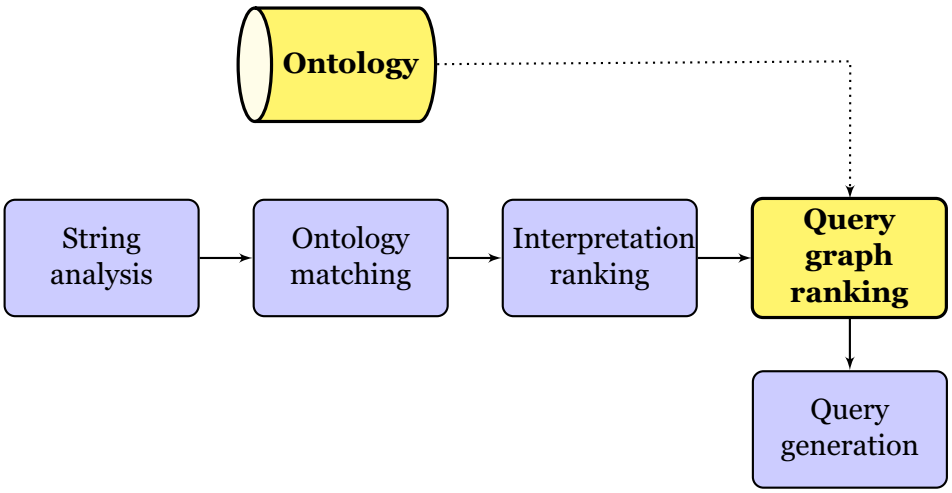



Figure 9.1: System architecture overview: Query graph ranking.

## 9.2 Implementation

The graph rank receives a Combined data structure from the interpretation rank. Combined contains three combinations of objects where each combination represents the user's input question. The content of the Combined data structure is illustrated in Tab. 9.1. Here we can see the three sentences, phrase by phrase, followed by the concept the phrase is

| | | | |
|---|---|---|---|
| 1 | Current field operator | of | Ekofisk |
| | Field | Reserve | FieldReserve |
| | Company | reservesResourceClass | name |
| | currentfieldOperator | null | null |
| | 2.0 | 1.0 | 0.0 |
| 2 | Current field operator | of | Ekofisk |
| | Agent | Reserve | Field |
| | Field | reservesResourceClass | name |
| | ˆcurrentfieldOperator | null | null |
| | 3.0 | 1.0 | 0.0 |
| 3 | Current field operator | of | Ekofisk |
| | Company | Reserve | Reserve |
| | Field | reservesResourceClass | name |
| | ˆcurrentfieldOperator | null | null |
| | 3.0 | 1.0 | 0.0 |

Table 9.1: Query generation input.

related to, the attribute type or the concepts relation target, the name of the relation and finally the phrase's edit distance to its retrieved ontology object. The column will contain "null" if the object type does not contain some of this information. As we can see the three combinations of objects are fairly similar, which is normal, because the phrases with the lowest scores often are repeated in different places in the ontology, often with similar names. There are relations with the same name or facets, typically names, which are represented in multiple concepts, like "Statoil Petroleum AS" being both a Company name and an Agent name, like FieldLicensee and TUFacility. Also words like "operator" and "facility" are parts of many different concepts. It would not have been necessary to have the $N$-best phrase combinations as input if the system only used the best one. Therefore we once again make use of the ontology, illustrated in Fig. 10.1, and employ how the different concepts are related [5] to each other in order to get a final opportunity to determine which objects to use. By doing this we can locate the ontology objects which is connected to each other in the graph. This works as the final sentence disambiguation when objects with no connection to the other objects will be removed.

### 9.2.1 Concept neighbors

To be able to select the correct objects at this stage we take another factor into account: in addition to the string similarity score of the input phrases' edit distance to the ontology object names. We are now also interested in how the potential objects are related to each other, or more precisely, how close an object is to the other objects in the graph, which is our ontology. Doing this is complicated and we have to figure out how to scale these two measurements against each other. One way to handle the scaling is to find out how close the objects that represent the different parts of the sentence

| | | |
|---|---|---|
| 1 | Statoil Petroleum AS | Current field operator |
| | Agent | Field |
| | name | Company |
| | null | currentfieldOperator |
| | 0.0 | 2.0 |
| 2 | Statoil Petroleum AS | Current field operator |
| | Company | Agent |
| | name | Field |
| | null | ^currentfieldOperator |
| | 0.0 | 3.0 |
| 3 | Statoil Petroleum AS | Current field operator |
| | Company | Company |
| | shortName | Field |
| | null | ^currentfieldOperator |
| | 0.0 | 3.0 |

Table 9.2: Example input.

are to the objects representing the other parts of the sentence and score each part with a number according to how many edges it is between each object an its closest neighbor. We will then have one score ranking the objects by edit distance and one ranking their relevance through similarity in the graph, and finally put weights on the two measurements and then calculate which objects best represent the initial input sentence. This idea is still under development and is not yet implemented in the system. There is, however, implemented a simplification of this neighbor-theory as a temporary solution in the graph ranking. Here the idea is to check if the second object is a neighbor of the first object in the first answer set. If it is, we add it to the query generator, if not, we check the object in the second set to see if the object related to the same phrase here are closer to the first object than the second object in the first answer set. If it is we add this to the query generator, if not we carry on and check all the $N$ objects in the $M$ answer sets until we have the $N$ objects that is the closest according to the ontology graph. This method makes the dangerous assumption that all the objects in the $M$ answer sets are equally correct, and that the closest one is the correct one, no matter what. This is one of the reasons for having such a small answer set, just 3 combinations of objects, so that the edit distance of all the objects is fairly similar.

In Fig. 9.2 is a simplified sketch of how the retrieved ontology objects can be represented in the ontology graph. In this example the circles represent concepts, the squares represent attributes and the lines represent relations. The part in red represents the objects that have been retrieved by the system, “Atr1”, “Atr.2” and *operator*. In this example the system has retrieved two attributes, the one in the top of the graph, “Atr2” and the one in the bottom of the graph, “Atr1”. The system has also retrieved the relation *operator*, where three different findings for the same phrase are represented in the graph, as relations connected to concept A. Without

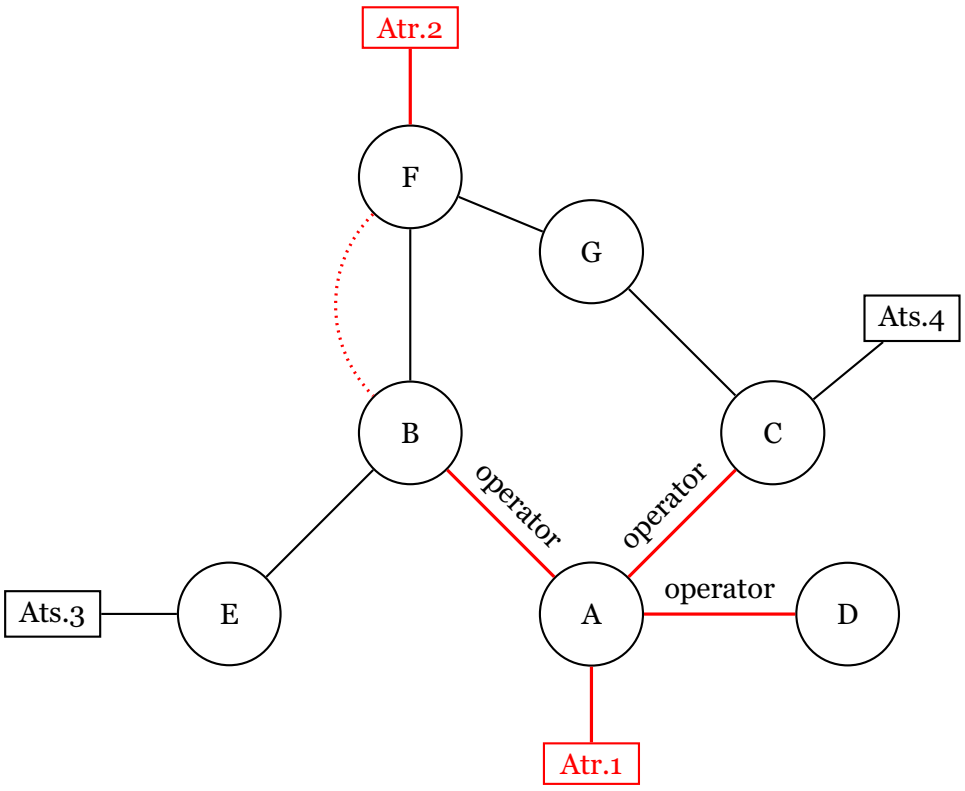



Figure 9.2: Example of shortest path in graph.

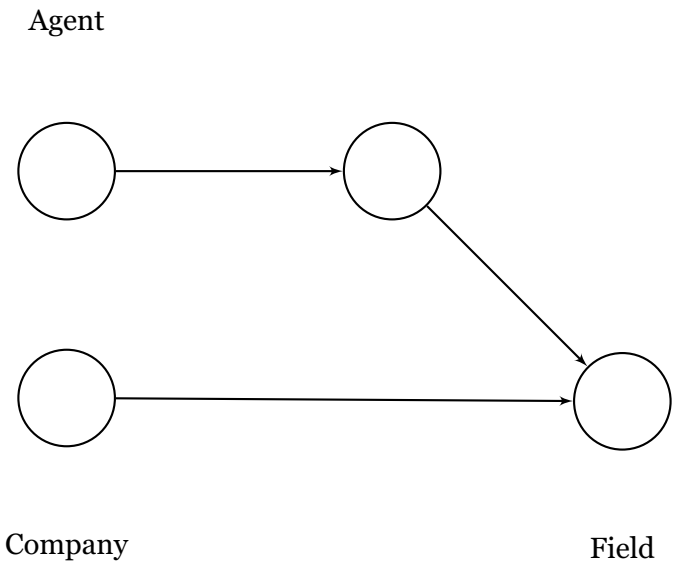



Figure 9.3: selection of the ontology represented as a graph.

looking at how the nodes are connected in the ontology it would have been arbitrary which one of the three *operator* relations that would have been chosen. However, with the knowledge of what nodes each node is related to we can conclude that the *operator* relation to the right, relating A to D, most likely is irrelevant since it is impossible to connect B with the other attribute retrieved, "Atr.2". By counting how many steps there are between the retrieved nodes we can see that by taking the path up and left from A, through B to F where the dotted line is drawn, we get the shortest path between the retrieved nodes, which also is a connected graph. This way of using the ontology to create a graph employing our retrieved ontology objects and adding the concepts from the ontology which creates a complete subgraph is inspired by the SINA [15]-system.

Assume the input sentence "where is Statoil petroleum AS current field operator". If we remove stopwords and the system runs the substring method, the input for the query generator is as shown in Tab. 9.2. The problem in this example is that the first object found for "Statoil Petroleum AS" is a facet related to the concept "Agent", whereas the facet we are interested in, in this specific case, is "Company". In this example the method for getting the nearest neighbors from the top of each column will solve our problem. From Fig. 9.3 we can see an estimation of how the ontology graph looks for the mentioned concepts. *Agent* has another concept linking it with *Field*, and *Company* has a direct link with Field. I have left out the relation names as they are irrelevant for this example, there are also several relations between Field and Company. In a case like this our system would note that Agent and Field have one node separating them, and move on to the next possibility which is Company and Field. As it is shown these concepts are neighbors and the system will select these objects since they are the ones with the highest ranking and all objects

are neighbors, so there is no point in moving on. It will send an attribute object with a name type and a relation object between the concepts Field and Company to the query generator.

## 9.3 Future work

As mentioned earlier the neighbor method needs to weight the edit distance and how many steps one node is from the other. The current method is naive and rests on the assumption that the closest of the neighbors always is the correct one. There are also the case of several nodes with more than one connection, and the graph is much more complicated, and it is a lot more relations, than the previous figures show. This implies that it often will occur that all the concepts chosen have the same connection distance, and counting the distance won't give much additional information. It is in fact the case that there seems to be some central nodes, Field, Facility, Company and Operator, that connects with many nodes, and have a lot of connections between each other. Because of this the connect distance will rarely be more than two, and most often one.

To improve the query graph ranker it is imperative that we implement the previous described idea. Because the naive current implementation is unsound, and will easily fail to select the correct ontology objects.

# Chapter 10

# Query Generation

## 10.1 Overview

The fifth and final part of the OptiqueNLQF system is the query generation. This is where all the information retrieved in earlier stages is transformed into a formal database query. The query generation's input is the Combined data structure introduced in the previous system part. At this stage we facilitate the generation of the query in the correct manner. OptiqueNLQF has run its course when the query is generated, but it will communicate with the Optique back-end and execute the query using the already existing software.

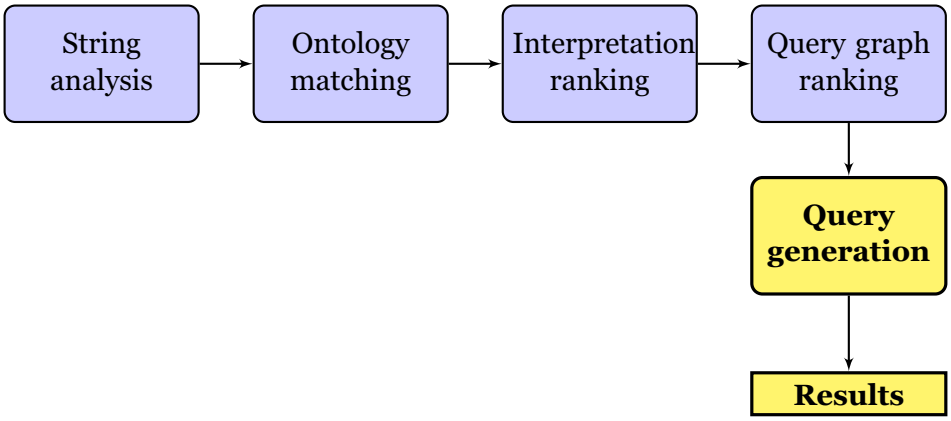



Figure 10.1: System architecture overview: Query generation.

### 10.1.1 Query language

The query language currently supported by OptiqueNLQF is SPARQL [1]. The goal for the system is to implement a query visitor [3] and give the user the opportunity to select between several query languages, like the Structured Query Language (SQL), JSON and SPARQL. This had to be postponed due to lack of time and a SPARQL query generator was implemented, mostly because this is the query language we were most familiar with. The

[1] http://en.wikipedia.org/wiki/SPARQL

| Statoil Petroleum AS | Current field operator |
|---|---|
| Company | Field |
| name | Company |
| null | currentfieldOperator |
| 0.0 | 2.0 |

Table 10.1: Example input.

OptiqueVQS system uses JSON queries[2], but it can also execute SPARQL queries. In fact, it translates every information request into SPARQL and gives the user the option to see the information request represented as a SPARQL query before the identical JSON query is executed on the system.

## 10.2 Design choices / Implementation

The objective for the query generator is implementing a generic generator that handles the objects inputted and generates the final query as a string. The idea behind our implementation is to identify what type of ontology objects we are dealing with as they are traversed and use a series of tests to handle the various object types and generate the query in the correct SPARQL notation.

### 10.2.1 Writing the query

At this stage in the workflow of the system all the results have been selected and it is just writing the query that remains. We operate with a set of connected objects and the main difficulty here is to make a generic method that will create the correct triple from the given objects. The query generation part of the system does not contain the most complex algorithms or complicated programming parts, but there are still a lot to calculate and a lot that must be taken into account when generating the SPARQL query. In our current implementation we first check what type of ontology object the current ontology object is, this is determined by looking at the data structure Combined. If row three contains something (see Tab. 10.1) we know the object is a relation. If the third column is null, but the second one contains something we know it is an attribute, and if none of the above checks out the ontology object is a concept. From this we want to generate triples where we have a subject, a predicate and an object. We also need to keep track of different variables and literals, for which we have declared two counters, literal (?l{literal counter}) and variable (?v{variable counter}). When a concept object is encountered we generate a triple where the ?{variable number} is the subject, ns1:type, or simply an *a*, is the predicate and a concept uri is the object. “Company” will generate the following triple:

?v1 a http : //sws.ifi.uio.no/vocab/npd – v2#Company .

[2]http://www.jsoniq.org/

?v1 ns1 : typehttp : //sws.ifi.uio.no/vocab/npd – v2#Company .

The two triples above will be handles as identical triples. We keep a track of the concept name and variable number, since it will occur later and we need to combine it with additional information, and we increment the variable counter. When the ontology object input is a relation we have to read the concept, the concept target and the relation name. Unless one of the concepts is already in use, this will generate three lines of SPARQL code. The first two lines will be generated with the same structure as the concept object. The relation will be written as the predicate between the two concepts, which will be represented by its appointed variables. The relation *current field operator* will generate the following triples:

?v1 a http : //sws.ifi.uio.no/vocab/npd – v2#Field .

?v2 a http : //sws.ifi.uio.no/vocab/npd – v2#Company .

?v1 http : //sws.ifi.uio.no/vocab/npd – v2#currentFieldOperator ?v2 .

Note that the variable counter is increased to two after the first concept. The third object type is attributes, and these are a little more complicated than the other two. With attributes we have to connect a type to a concept and add a filter at the end of the query. To do this we use the attribute type to write the filter correctly. If the attribute is of the type string we add a filter function called regex (regular expression) and uses the string found in the ontology as its argument. The system is not complete at this area and handles other facet types, like numbers or intervals, poorly. Given the ontology facet object “Statoil Petroleum AS” the system would generate the following SPARQL triples:

?v1 a http : //sws.ifi.uio.no/vocab/npd – v2#Company .

?v1 http : //sws.ifi.uio.no/vocab/npd – v2#name ?v1 .

Filter(regex(?v1, “Statoil Petroleum AS″))

The triple with the filter is added to a queryFilter() method and will be written last in the SPARQL query. If there are more than one attribute object there will be added more filters, the literal counter then works in the same way as the variable counter. Using these rules the input information request “where are Statoil Petroleum AS the current field operator?” the following SPARQL query will be generated:

```
PREFIX ns1:
        <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2\#>
SELECT DISTINCT *
WHERE {
?v1 ns1:type ns2:http://sws.ifi.uio.no/vocab/npd-v2\
        #Field .
?v2 ns1:type ns2:http://sws.ifi.uio.no/vocab/npd-v2\
        #Company .
?v1 ns2:http://sws.ifi.uio.no/vocab/npd-v2\
        #currentFieldOperator ?v2 .
?v2 ns2:http://sws.ifi.uio.no/vocab/npd-v2\#name ?l1.
Filter(regex (?l1, ``Statoil Petroleum AS''))
}
```

Listing 10.1: SPARQL query

The "SELECT", "WHERE" and the curly brackets are added before and after the triples are automatically added in the start and at the end. For now the query is always written with the "DISTINCT" modifier and a *, the star meaning "select everything from the following". The prefix *<http://sws.ifi.uio.no/vocab/npd-v2#>* used is from the OptiqueVQS [3] and represents all the ontology objects. The other prefix *PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>* is the standard RDF syntax for SPARQL queries.

## 10.3 Further work

This particular part of the system needs a lot of changes to be able to generate a larger variety of queries. A necessary improvement is adding the possibility to filter on attribute types other than strings. The "FILTER (regex (?l{literal number} "attribute name"))" is, as of now, the only supported alternative. The scope of the system's capability would greatly increase if OptiqueNLQF was able to formulate queries given an information request like:

Fields with remaining oil more than 100 Million m3.

Transportation and Utilization Facilities valid from 4 – 5/2015.

The first information request requires the system to possess the ability to recognize values and value types, which in this case is "100 million" and "cubic meters" (m3). The system also has to identify textual representations of numbers (in this case "million") and calculate this according to any possible numeric number in the input. The first example would return 100 000 000 (one hundred million). This number is then used as a filter in the query with the corresponding measurement, in this case cubic meters (m3).

[3] http://fact-pages.fluidops.net/resource/VisualQueryFormulation

The system also has to observe the adjective “more” to add a *greater than* (≥) attribute to the SPARQL filter. We also have to connect the filter literal to the correct concept, in this case RemainingOil. The second example sentence listed above contains a date which is a common datatype property in the ontology. handling dates is complicated since the phrase containing the date can return, practically, any concept from the ontology which has a date attribute. To handle cases where the input phrase contains a date, we have to implement a method for checking if any of the selected concepts have a date as its attribute type. Then it will have to formulate a SPARQL filter containing the date, accordingly.

# Chapter 11

# Evaluation

## 11.1 Testing

Even though OptiqueNLQF is far from finished and this is only an initial prototype, the system needs some sort of evaluation to test its potential and its limitations. Hopefully the evaluation will unravel the system's most pressing needs for improvement and shed light on the system's flaws. Since it is so early in the project, and the system is unable to handle a wide enough variety of sentences, the testing will be performed on a test set developed by us. The purpose of the testing will be to test the system's limits and therefore the test set will contain example sentences in increasing complexity and varying in sentence length and word order. The test results will also be evaluated in a more lenient way, and the system will be judged by its ability to generate sensible SPARQL queries, and not what results are retrieved by the Optique back-end system. The system will run the tests with the parse method and the substring method, with both the stopwords action and the case sensitivity activated and deactivated. The system will also be tested on how fast all the queries are generated. The tests will primarily be executed on the NPD annotated ontology we have used during the development of OptiqueNLQF. We will also perform some tests on another ontology called Northwind [1]. This ontology is also developed as a part of the Optique project, but unlike the NPD ontology describing the Norwegian Petroleum Directorate, Northwind describes the database of a company. We want to test our system on another ontology to test the system's ability to work on a new domain, and see if it occurs any problems when switching to an unknown ontology.

### 11.1.1 Test data

The data used in the following tests from the NPD ontology [2] will mainly consist of the concepts: Company, Field, Wellbore, Facility and Operator. These concepts have many relations. We will primarily employ *currentFieldOperator*, *developmentWellboreForLicence*, *drillingOperat-*

[1] optique-northwind.fluidops.net/
[2] http://sws.ifi.uio.no/project/npd-v2/

*orCompany* and *pipelineOperator*, other relations will occur, these are relations between the concepts earlier listed. The most frequent attributes are “Statoil Petroleum AS” and “JACKET 4 LEGS”, these are a company and a facility type respectively. The reason for using these is because they are the most frequent in the database and will therefore yield results for the majority of the information requests.

### 11.1.2 Test set

To achieve a broad evaluation and produce a sort of measuring scale for the sentence complexity we generate increasingly harder sentences, starting at a simple concept name, increasing until the system’s starts failing. The system also generates a lot better queries if the input sentence contains as few other words additional to the words or phrases representing an ontology object.

#### Simple test input

The first simple test phrases:

- Fields
- All fields
- Give me all fields
- Return all fields

These phrases practically means the same thing, but will possibly generate different queries. We will therefore test the system knowing that fewer words, and words more directed to the ontology, will generate better queries. After the tests with one concept from the ontology we will increase the complexity and test two ontology concepts. At this stage there are two options, the concepts are related in the ontology, and the concepts are not related. The test phrases with two concepts are:

- Field company
- Field facility
- Company facility

In the first two examples above the concepts are related, but in the third example they are not. This will test the system’s ability to generate relations between concepts, and see what happens when two ontology concepts are located which do not have a relation. The next step is testing how the system handles the provided relations. Relations are often several words put together, hence the user needs to formulate the input phrase correctly according to the ontology. The same applies to the attributes, which in this case will be counted in the same category as relations. We have generated the following phrases:

- status for field
- Pipeline operator
- Statoil Petroleum AS

These initial examples are the most basic tests we will perform and they will work as tests to see if the different settings return the same answers given the most basic input.

### Complex test input

Increasing the complexity of the input sentence further involves adding one ontology object of a different type to any of the above examples. Also making the input phrase more authentic by adding words and forming a continuous words sequence. To illustrate this, the following phrases are constructed:

- Fields operated by Statoil Petroleum AS
- Current field operator of Statoil Petroleum AS
- Fields where the operator is Statoil Petroleum AS
- companies with jacket 4 legs facilities
- Development wellbore for license with Statoil Petroleum AS
- Field where drilling operator company is Statoil Petroleum AS

These six sentences contain concrete ontology objects and some other words to make the sentences more realistic. These input phrases will test the systems ability to handle two to three objects, and objects of varying types. The words in the sentence still match the ontology object names, and words forming a relation name are put in the order matching the relation object name.

### Technical information

The tests for OptiqueNLQF will be performed on an Asus k55vm notebook [3] with an Intel Core i7 processor, 8GB RAM and an Intel 520 180GB SSD. The Java version used during the testing is the jre 7 Standard VM, the program is run using Eclipse Juno [4].

## 11.2 Result

The results discussed in this section are based on the phrases presented in Sec.11.1.2, and the phrases' generated SPARQL queries are located in the Appendix Chap.13.

[3] http://www.asus.com/Notebooks_Ultrabooks/K55VM/
[4] https://eclipse.org/juno/

### 11.2.1 Simple test results

In Tab. 13.1 and Tab. 13.2 we see the results from our initial tests. These test were run using the parse-tree method and the substring method respectively. As shown in the tables, the methods gave identical results. These particular tests are run with the removeStopwords() and ignorecase() methods deactivated. This however have no impact on these tests and every setting retrieves the same result. We can nonetheless conclude that the system will return the correct SPARQL query given a vary basic input phrase that is matched with the ontology object names. If we direct our attention to the rightmost column in the earlier mentioned tables we can see the time use for the query generation. From the data currently retrieved we can see that the parse-tree results ranges from 6.2 seconds to 7.6 seconds, while the substring method ranges from 0.7 seconds to 1.6 seconds. For both tests the time is clearly correlating with the word count. The parse-tree time increases more than the substring time, but as we can see the substring time more than doubles its time use from the one-word test to the three-word test. It is too early to draw any conclusion, but the numbers are clear, nonetheless. Also note that our previous assumption that the time use of the reading of the language model seems to be fairly accurate, at least for the time being.

### 11.2.2 Complex test results

Tab. 13.3 shows the SPARQL query generated from the input phrase "Fields where the operator is Statoil Petroleum AS". This is clearly an incorrect query and we can start of by stating that running the system without the stopword removal action will generate some wild queries when a complete sentence is provided. Because of this we see no reason to continue testing without the stopword removal.

Tab. 13.4 and Tab. 13.5 contains the six phrases presented in the previous section, SPARQL queries and the systems run time to generate the query. The tables show the system performance with the parse-tree method and the substring method respectively. The first three test phrases are:

- Fields operated by Statoil Petroleum AS
- Current field operator of Statoil Petroleum AS
- Fields where the operator is Statoil Petroleum AS

All these phrases have the same basic meaning, and generate very similar SPARQL queries for both the parse-tree and substring method. The second phrase is the one generating the most correct query, and this one will retrieve the correct answers from the Optique system. In the other two queries the triples concerning the Operator and the *operatorForField* are redundant and will cause a "Nothing found" data return. Any of these phrases will be notably affected by the status of the ignorecase option, the same queries will be generated regardless of run configuration. The time use for creating these queries are still in favor of the substring method,

although the time difference seems to be decreasing. The time difference is 4.4 seconds for the first query, and just under 3 seconds for the other two.

The fourth example phrase "Companies with jacket 4 legs facilities" generate quite different queries given the different system options. The idea of this information request is to retrieve all companies that have facilities of the type JACKET 4 LEGS. As sees in the tables earlier mentioned, as well as Tab. 13.6, the queries generated are very different. The parse-tree method define two concepts, Company and Jacket4LegsFacility, while the substring method define Company and Jacket4LegsFacility, connects the Jacket4LegsFacility to an attribute "JACKET 4 LEGS", defines the concept Facility and relates it to Company through *pipieLineOperator*. In this case it is the parse-tree method that creates the correct query, which is interesting, since it is the substring method that found the lowest scored phrase combination. The reason for this it that the substring method located the Company concept, most likely with a edit distance of 0 or 1, the JACKET 4 LEGS attribute, also with a very low score, and connected this attribute to Company using the Jacket4LegsFacility concept and the *facilityType* relation. Finally it takes the Facility concepts and connects this to the Company concept with the *pipelineOperator* relation. This being the lowest combination of objects is implicit. However, when the parse-tree method creates its parse tree it does not recognize the "JACKET 4 LEGS" sub-phrase. Therefore it is left with the two concepts: Company and Jacket4LegsFacility, which is not connected in the ontology, and therefore no relation is added. This sentence's generated query executed using the parse-tree method with activated case sensitivity is presented in Tab. 13.6. It illustrates some serious flaws with case sensitivity. Because of the parse-tree method being unable to retrieve the "JACKET 4 LEGS" phrase and the Jacket4LegsFacility concepts is in lowercase letters the objects retrieved with the lowest scores are some random attributes and therefore the query generated is gibberish.

The fourth example sentence supports the assumption that the sub-string method will find the correct ontology objects, while the parse-tree method will struggle with relation names containing multiple words. In this sentence the relation name *developmentWellboreForLicence* was the key for generating the correct query, and the parse-tree was unable to locate this particular sub-phrase. The substring method's generated query however works perfectly and is about 2.3 seconds faster. The final input sentence was supposed to illustrate the same problem, but instead display another one. The relation *drillingOperatorCompany* was the key in this example. It was not retrieved by any of the system methods, but not because they were unable to locate it. The reason is in fact that the words by themselves acquire a lower combined score. The concepts Operator and Company and a *taskType* attribute "drilling" all get a score of 0 in ignorecase mode and a score of 1 in case sensitivity mode. The relation will retrieve the same scores for the words, but an automatic +2 total score because of the spaces between the words. This example additionally presents another interesting discovery: the time use of the substring method exceeds the time

use of the parse-tree method. The parse-tree method is in fact about one second faster.

### Testing the system's time use

To test the system's time use we will create a set of sentences with word count ranging from 8 to 10 words, containing phrases from the ontology, full sentences based on the ontology objects and some independent test sentences. The sentences are created in the manner such that the first phrase in the 8-word sentences corresponds to the first phrase in the 9th and 10th word sentences, as does the second phrase and the third phrase. The phrases are:

**8 word sentences:**

1. Company field facility wellbore well point wellhead area
2. The company with the most facilities and wells
3. The table on the floor is very big

**9 word sentences:**

1. Company field facility wellbore well point wellhead area discovery
2. The company with the most facilities and oil wells
3. The table on the floor is big and blue

**10 word sentences:**

1. Company field facility wellbore well point wellhead area discovery Refining
2. The company with the most loading facilities and oil wells
3. The table on the floor is big and mostly wood

In Tab. 11.1 we can see the sentences presented above as sentence 1, 2 and 3, containing 8, 9 and 10 words, respectively, for both the parse-tree method and the substring method. We can see that the substring method is the fastest for all three sentences in their eight word version. When one word is added to the sentences things change, the parse-tree method now run sentence 1 the fastest. In all ten word sentence examples the parse-tree method is plainly faster. At nine words the parse-tree method produces 16 phrases, and the substring method produces 45, and the difference begins to grow. It is clear that the parse-tree method makes up for the time it spends on loading the language model after about 9-10 words, and that the decrease in phrases pays of. In Fig. 11.2.2 we can see how the red lines, representing the three sentences generated using the substring method, have a much steeper increase compared to the blue lines, the sentences generated using the parse-tree method.

| Sentence | words | parse-tree() | substring() |
|---|---|---|---|
| 1 | 8 | 10.045 | 8.785 |
| | 9 | 10.727 | 11.466 |
| | 10 | 11.569 | 15.340 |
| 2 | 8 | 9.885 | 7.830 |
| | 9 | 10.267 | 9.810 |
| | 10 | 11.397 | 12.850 |
| 3 | 8 | 10.087 | 6.780 |
| | 9 | 10.286 | 8.720 |
| | 10 | 10.909 | 10.930 |

Table 11.1: Time table for query generation.

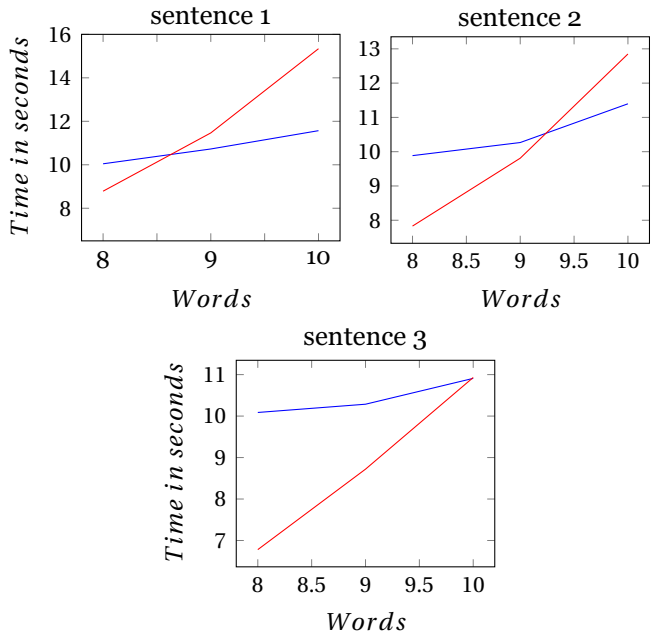
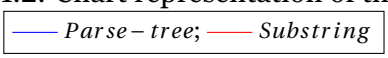



Table 11.2: Chart representation of time use.



## 11.3 Northwind

To test the Northwind ontology we will use some simple phrases containing the concept and relation names from the ontology. This will resemble the initial test 11.1.2 performed on the NPD ontology. Only this time, we are not as interested in whether the queries generated are correct, as we are in how seamless the transition from one ontology to another will be. There are not yet added any attributes to the ontology, so our test phrases will only contain concepts and relations. The following phrases are tested with the Northwind ontology as the OptiqueNLQF ontology input:

1. Which employee has this territory.
2. The shipped data of the order.
3. Location of company.

The results for these phrases are displayed Tab. 14.1, for the parse-tree method, and Tab. 14.2, for the substring method, in appendix B 14. We have not given the Northwind ontology much attention and we have not gone in depth to understand the structure of the ontology. It is neither as complicated nor as complete as the NPD ontology, so to run more tests on Northwind is not necessary at this time. The transition to another ontology was however a success, and the system had no problems creating the queries shown in Appendix B 14. This is much because of the system's back-end ontology parsing part's ability to extracted the ontology objects correctly. The main problem with new ontologies at this stage is to learn the names of the concepts and relations and learn the structure of the ontology. This is time consuming for new users, and it also oppose one of the focus points of this project; creating a system which is easy to use for a person with limited knowledge of databases and query languages.

## 11.4 Summary

OptiqueNLQF has proven to generate functioning queries within a limited scope of complexity. The two different methods of running the system, parse-tee() and substring(), have proven to provide quite similar queries, even though the substring method works better when the input gets more complex. The stopword remover have demonstrated it's importance as the SPARQL query in Tab. 13.3 shows. The case sensitivity test results did not provide any clear answers. The problem being that a search for "jacket 4 legs" will possibly yield an attribute name, "JACKET 4 LEGS", or a concept name, Jacket4LegsFacility, depending on what setting the case sensitivity is set to. This issue needs some more testing before we can present a conclusion. The system has also confirmed that it can handle the process of working with a new ontology, when OptiqueNLQF was tested with the Northwind ontology. It was a welcome result, as this was one of the projects goals.

It would have been preferable to perform some sort of user test for a more conclusive system evaluation. However, this turned out to pose several challenges, as the user would need extended knowledge of the system in order to be able to test it in a sufficient manner.

# Part III

# Conclusion

# Chapter 12

# Conclusion

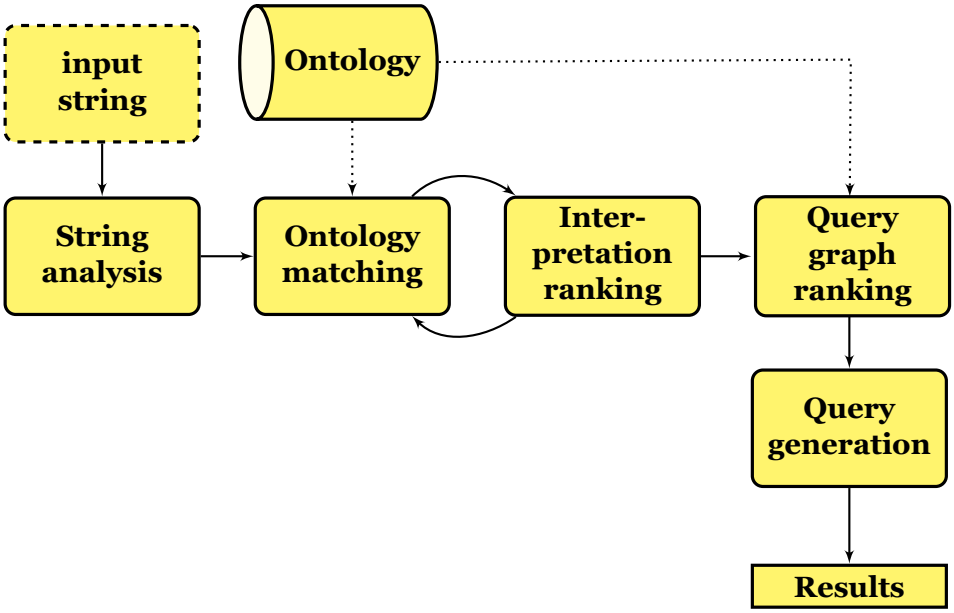



Figure 12.1: System architecture overview.

## 12.1 Overview

In this thesis the development and functionalities of the OptiqueNLQF system, a prototype of a natural language query formulation system, has been presented. The system architecture and its main parts have been described. A final illustration of system workflow is presented in Fig. 12.1.

### 12.1.1 System comparison

In Chap. 2, the chapter describing four similar systems, a table scoring the systems was presented. This table is represented again in Tab. 12.1, only this time the scores of the OptiqueNLQF are included. Let it be stated that this scoring is done by the creators of OptiqueNLQF, and is in no way

Table 12.1: Projects

| | SINA | USI | Pythia | Quepy | OptiqueNLQF |
|---|---|---|---|---|---|
| Setting up a new project | ✗ | ✗✗ | ✗ | ✓✓ | ✓ |
| Use on new domain | ✓ | ✗ | ✓ | ✓ | ✓✓ |
| Scalability | ✓ | ✓ | ✓✓ | ✗✗ | ✓/✗ |
| Upper limit | ✓ | ✓✓ | ✓ | ✗ | ✗✗ |
| Open source | ? | ✗ | ? | ✓ | ✗ |

This table shows a visualization of the differences of the systems. We have used a grading-system with the scale: ✗✗,✗, ✓, ✓✓.

an official ranking or evaluation. The projects are merely defined by the descriptions given in the papers in which they are presented.

- Setting up a new project is easy and the system will be functional from the beginning. The need for adding synonyms and similar words to the ontology objects in the future will however increase the workload before the first time the system can be used.
- As shown in the result chapter, tested with the Nortwind ontology, OptiqueNLQF is easily used on new domains, and this is done simply by providing a new ontology for parsing.
- As scalability is concerned the system needs some improvement. It does nonetheless make sense of different types of ontology objects, and, with its limitations, handles this fairly good.
- The system's upper limit is, for the time being, very low. It works fairly well when the input contains two, maybe three, ontology objects. Any more than this and the queries generated are nonsensical.

The tests of the system demonstrated that there is potential, and for expressing short, direct information needs the system is able to provide the correct query formulation. Regarding the system, the two finishing parts, the query graph ranking and the query generation, have a lot of potential for improvement. Improving how the system handles ontology objects as neighbors in the ontology graph before starting the query generator can change the system's capability drastically. The parse-tree have shown its potential in phrase extraction and performing some improvements on this part as well can help develop the competence of OptiqueNLQF.

## 12.2 Future work

Some other ideas, apart from improving the already created system parts, would also be interesting to implement in OptiqueNLQF. The fact that the users would have to know the exact names of the ontology objects to formulate the correct queries is a huge and quite obvious disadvantage. An improvement in this area is highly needed, and there are several ways in

which this can be done. One way is to add a set of synonyms as a part of the ontology object information. This idea was introduced in the USI-system [20] as an added part of the ontology. The ontology matching would in this case compare the input phrases to the ontology object names, and its synonym list. This could work for concepts, but it would require a lot of varying phrases for the relations. It would likely add resembling phrases and descriptions for the different relation names, which have potential to be similar enough as it is. This could further increase the possibility of extracting an unintended ontology object, if the user's input string fails to be precis enough. Another problem faced if we intend to let the input string structure be as free as possible is the word order in the relation names. We want the relation *currentFieldOperator* and an input string such as "Fields currently operated" to have an as low score as possible. To enable this we could implement a method for rearranging the word order, after finding the lemma [1] of the phrases. Regardless, implementing the lammatization method from the OpenNLP toolkit would be an interesting notion.

An idea for parsing the words in the input string in a more suitable way, according to the ontology, would be to implement the ontology into the parser's language model. This idea is inspired by the Pythia-system [19], where the input sentence is analyzed by an ontology-based grammar. We would create a more domain specific language model. The sentence parsing would automatically recognize the ontology object names and identify these as a part of the sentence. By doing this we could perform the string analysis and the correct phrase extraction in the same operation. This could also improve the correct phrase extraction rate of the parse-tree method.

It is also of interest to change the structure of the system and let the user see the final query both in SPARQL language and as a graph in the OptiqueVQS system before the user chooses to execute the query. At this stage we want to give the user the opportunity to alter the query or add additional information. This is one of the main future goals for the continuing expansion of OptiqueNLQF.

[1] http://en.wikipedia.org/wiki/Lemmatisation

# Chapter 13

# Appendix A

The prefix used in all the following SPARQL queries:

- PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
- PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>

The *ns1* prefix is the standard RDF prefix for SPARQL queries. The *ns2* prefix maps to the ontology objects found in the NDP ontology.

## 13.1 Initial tests

Table 13.1: OptiqueNLQF phrase test with parse-tree method.

| Input phrase: | Query | Run time |
|---|---|---|
| Fields: | **SELECT** * <br> **WHERE** { <br> ?x1 **ns1**:type **ns2**:Field . <br> } | Time: 6.157 |
| Field comany: | **SELECT** * <br> **WHERE** { <br> ?x1 **ns1**:type **ns2**:Field . <br> ?x2 **ns1**:type **ns2**:Company . <br> ?x1 **ns2**:currentFieldOperator ?x2 . <br> } | Time: 6.647 |
| company facility: | **SELECT** * <br> **WHERE** { <br> ?x1 **ns1**:type **ns2**:Facility . <br> ?x2 **ns1**:type **ns2**:Company . <br> ?x1 **ns2**:pipelineOperator ?x2 . <br> } | Time: 6.789 |

| | | |
|---|---|---|
| Field facility: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Facility .<br>} | Time: 6.807 |
| Status for field: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:FieldStatus .<br>?x2 **ns1**:type **ns2**:Field .<br>?x1 **ns2**:statusForField ?x2 .<br>} | Time: 7.547 |
| Pipeline operator: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:FeederPipeline .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:pipelineOperator ?x2 .<br>} | Time: 6.882 |
| Statoil Petroleum AS | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 7.095 |

Table 13.2: OptiqueNLQF phrase test with substring method.

| Input phrase: | Query | Run time (sec) |
|---|---|---|
| Fields: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>} | Time: 0.748 |
| Field comany: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:currentFieldOperator ?x2.<br>} | Time: 1.206 |

| | | |
|---|---|---|
| company facility: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Facility .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:pipelineOperator ?x2 .<br>} | Time: 1.264 |
| Field facility: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Facility .<br>} | Time: 1.170 |
| Status for field: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:FieldStatus .<br>?x2 **ns1**:type **ns2**:Field .<br>?x1 **ns2**:statusForField ?x2 .<br>} | Time: 1.624 |
| Pipeline operator: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:FeederPipeline .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:pipelineOperator ?x2 .<br>} | Time: 1.217 |
| Statoil Petroleum AS | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 1.720 |

## 13.2 Second evaluation

Table 13.3: Test without stopword removal.

| Input phrase: | Query | Run time |
|---|---|---|
| fields were the operator is Statoil Petroleum AS: | **SELECT** * **WHERE** { ?x1 **ns1**:type **ns2**:Field . ?x2 **ns1**:type **ns2**:Award . ?x3 **ns1**:type **ns2**:WellboreDocument . ?x3 **ns2**:documentFormat ?v1 . ?x4 **ns1**:type **ns2**:Operator . ?x3 **ns2**:operatorForField ?x4 . ?x5 **ns1**:type **ns2**:Oil . ?x6 **ns1**:type **ns2**:Company . ?x5 **ns2**:^currentFieldOperator ?x6 . ?x6 **ns2**:name ?v2 . **Filter** (**regex** (?v1, "the")) **Filter** (**regex** (?v2, "Statoil␣Petroleum␣AS")) } | time: 10.701 |

Table 13.4: Tests with increased complexity, parse-tree method.

| Input phrase: | Query | Run time |
|---|---|---|
| Fields operated by Statoil Petroleum AS | **SELECT** * **WHERE** { ?x1 **ns1**:type **ns2**:Field . ?x2 **ns1**:type **ns2**:Operator . ?x1 **ns2**:operatorForField ?x2 . ?x3 **ns1**:type **ns2**:Company . ?x2 **ns2**:^currentFieldOperator ?x3 . ?x3 **ns2**:name ?v1 . **Filter** (**regex** (?v1, "Statoil␣Petroleum␣AS")) } | Time: 9.476 |

| | | |
|---|---|---|
| Current field operator of Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:currentFieldOperator ?x2 .<br>?x2 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 9.376 |
| Fields were the operator is Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Operator .<br>?x1 **ns2**:operatorForField ?x2 .<br>?x3 **ns1**:type **ns2**:Company .<br>?x2 **ns2**:^currentFieldOperator ?x3 .<br>?x3 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 11.336 |
| Companies with JACKET 4 LEGS facilities: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Company .<br>?x2 **ns1**:type<br>**ns2**:Jacket4LegsFacility .<br>} | Time: 8.975 |
| Development wellbore for licence with Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:DevelopmentWell .<br>?x2 **ns1**:type **ns2**:Wellbore .<br>?x3 **ns1**:type **ns2**:<br>WellboreDrillingMudSample .<br>?x2 **ns2**:^mudTestForWellbore ?x3 .<br>?x4 **ns1**:type **ns2**:Licensee .<br>?x5 **ns1**:type **ns2**:Company .<br>?x4 **ns2**:drillingOperatorCompany ?x5 .<br>?x5 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 11.660 |

| | | |
|---|---|---|
| Fields where the drilling operator company is Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Production–<br>LicenceWorkObligation .<br>?x2 **ns2**:taskType ?v1 .<br>?x3 **ns1**:type **ns2**:Operator .<br>?x2 **ns2**:operatorForField ?x3 .<br>?x4 **ns1**:type **ns2**:Company .<br>?x3 **ns2**:^currentFieldOperator ?x4 .<br>?x5 **ns1**:type **ns2**:Agent .<br>?x5 **ns2**:name ?v2 .<br>**Filter** (**regex** (?v1, "drilling"))<br>**Filter** (**regex**<br>(?v2, "Statoil␣Petroleum␣AS"))<br>} | Time:<br>13.117 |

Table 13.5: Tests with increased complexity, substring method.

| Input phrase: | Query | Run time |
|---|---|---|
| Fields operated by Statoil Petroleum AS | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Operator .<br>?x1 **ns2**:operatorForField ?x2 .<br>?x3 **ns1**:type **ns2**:Company .<br>?x2 **ns2**:^currentFieldOperator ?x3 .<br>?x3 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time:<br>5.066 |
| Current field operator of Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:currentFieldOperator ?x2 .<br>?x2 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time:<br>6.644 |

| | | |
|---|---|---|
| Fields were the operator is Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Operator .<br>?x1 **ns2**:operatorForField ?x2 .<br>?x3 **ns1**:type **ns2**:Company .<br>?x2 **ns2**:^currentFieldOperator ?x3 .<br>?x3 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 8.399 |
| Companies with JACKET 4 LEGS facilities: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Company .<br>?x2 **ns1**:type **ns2**:Jacket4LegsFacility .<br>?x2 **ns2**:facilityType ?v1 .<br>?x3 **ns1**:type **ns2**:Facility .<br>?x1 **ns2**:pipelineOperator ?x3 .<br>**Filter** (**regex**<br>(?v1, "JACKET␣4␣LEGS"))<br>} | Time: 4.705 |
| Development wellbore for licence with Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Wellbore .<br>?x2 **ns1**:type **ns2**:ProductionLicence .<br>?x1 **ns2**:developmentWellboreForLicence<br>?x2 .<br>?x3 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:drillingOperatorCompany ?x3 .<br>?x3 **ns2**:name ?v1 .<br>**Filter** (**regex**<br>(?v1, "Statoil␣Petroleum␣AS"))<br>} | Time: 9.230 |

| | | |
|---|---|---|
| Fields where the drilling operator company is Statoil Petroleum AS: | **SELECT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Field .<br>?x2 **ns1**:type **ns2**:Production–<br>LicenceWorkObligation .<br>?x2 **ns2**:taskType ?v1 .<br>?x3 **ns1**:type **ns2**:Operator .<br>?x3 **ns2**:operatorForField ?x1 .<br>?x4 **ns1**:type **ns2**:Company .<br>?x3 **ns2**:^currentFieldOperator ?x4 .<br>?x4 **ns2**:name ?v2 .<br>**Filter** (**regex** (?v1, "drilling"))<br>**Filter** (**regex**<br>(?v2, "Statoil␣Petroleum␣AS"))<br>} | Time:<br>14.293 |

Table 13.6: Case sensitive parse-tree method.

| Input phrase: | Query | Run time |
|---|---|---|
| Company with JACKET 4 LEGS facilities: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Company .<br>?x2 **ns1**:type **ns2**:BlowoutWellbore .<br>?x2 **ns2**:status ?v1 .<br>?x3 **ns1**:type **ns2**:Field .<br>?x2 **ns2**:currentFieldOperator ?x3 .<br>?x3 **ns2**:name ?v2 .<br>?x4 **ns1**:type **ns2**:Field .<br>?x3 **ns2**:currentFieldOperator ?x1 .<br>?x4 **ns2**:name ?v3 .<br>?x5 **ns1**:type **ns2**:Facility .<br>?x1 **ns2**:pipelineOperator ?x5 .<br>**Filter** (**regex** (?v1, "JACKET"))<br>**Filter** (**regex** (?v2, "4"))<br>**Filter** (**regex** (?v3, "LEGS"))<br>} | Time:<br>8.932 |

# Chapter 14

# Appendix B

The following SPARQL queries use the same prefixes as before. It is, however, not clear if the Northwind object are supposed to involve the *ns2* prefix affiliated to the NDP ontology. The queries still employ the *ns1* prefix.

Table 14.1: Tests on the Northwind ontology, parse-tree method.

| Input phrase: | Query | Run time |
|---|---|---|
| which employee has this terretory: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Region .<br>?x2 **ns1**:type **ns2**:Employees .<br>?x3 **ns1**:type **ns2**:Territories .<br>?x2 **ns2**:Employees_has_Territories ?x3 .<br>} | Time: 5632 |
| The shipped data of the order: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Shippers .<br>?x2 **ns1**:type **ns2**:Orders .<br>?x1 **ns2**:^ShipVia ?x2 .<br>?x4 **ns1**:type **ns2**:Shippers .<br>?x3 **ns2**:ShipVia ?x4 .<br>} | Time: 6106 |
| Location of company: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Location .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:Company.Location ?x2 .<br>} | Time: 5272 |

Table 14.2: Tests on the Northwind ontology, substring method.

| Input phrase: | Query | Run time |
|---|---|---|
| which employee has this terretory: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Region .<br>?x2 **ns1**:type **ns2**:Employees .<br>?x3 **ns1**:type **ns2**:Territories .<br>?x2 **ns2**:Employees_has_Territories ?x3 .<br>} | Time: 415 |
| The shipped data of the order: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Shippers .<br>?x2 **ns1**:type **ns2**:Orders .<br>?x1 **ns2**:ShipVia ?x2 .<br>} | Time: 727 |
| Location of company: | **SELECT DISTINCT** *<br>**WHERE** {<br>?x1 **ns1**:type **ns2**:Location .<br>?x2 **ns1**:type **ns2**:Company .<br>?x1 **ns2**:Company.Location ?x2 .<br>} | Time: 320 |

# Bibliography


[1] Ion Androutsopoulos, Graeme D Ritchie and Peter Thanisch. 'Natural language interfaces to databases–an introduction'. In: *Natural language engineering* 1.01 (1995), pp. 29–81.

[2] Bruce C Berndt and Ronald J Evans. 'The determination of Gauss sums'. In: *American mathematical society* 5.2 (1981).

[3] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[4] Bernardo Cuenca Grau et al. 'On Faceted Search over Knowledge Bases'. In: *Informal Proceedings of the 27th International Workshop on Description Logics, Vienna, Austria, July 17-20, 2014*. 2014, pp. 153–156.

[5] Peter Haase et al. 'Optique System: towards ontology and mapping management in OBDA solutions.' In: *WoDOOM*. 2013, pp. 21–32.

[6] Pascal Hitzler, Markus Krotzsch and Sebastian Rudolph. *Foundations of semantic web technologies*. CRC Press, 2011.

[7] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0130950696.

[8] Donald E Knuth, James H Morris Jr and Vaughan R Pratt. 'Fast pattern matching in strings'. In: *SIAM journal on computing* 6.2 (1977), pp. 323–350.

[9] L.D. Landau. 'The Viterbi Algorithm'. In: *Zhurnal Eksperimental'noi i Teoreticheskoi Fiziki* (1937), pp. 302–309.

[10] Ora Lassila and Ralph R Swick. 'Resource description framework (RDF) model and syntax specification'. In: (1999).

[11] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719, 9780521865715.

[12] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-13360-1.

[13] Deborah L McGuinness, Frank Van Harmelen et al. 'OWL web ontology language overview'. In: *W3C recommendation* 10.10 (2004), p. 2004.

[14] Natalya F Noy, Deborah L McGuinness et al. *Ontology development 101: A guide to creating your first ontology*. 2001.

[15] Saeedeh Shekarpour, A.C. Ngonga Ngomo and S Auer. 'Question answering on interlinked data'. In: *Proceedings of the 22nd international conference on World Wide Web* (2013), pp. 1145–1155.

[16] Martin G. Skjæveland and Espen H. Lian. 'Benefits of Publishing the Norwegian Petroleum Directorate's FactPages as Linked Open Data'. In: *Norsk informatikkonferanse (NIK 2013)*. Tapir, 2013.

[17] Ahmet Soylu et al. 'OptiqueVQS – Towards an Ontology-based Visual Query System for Big Data'. In: *International Conference on Management of Emergent Digital EcoSystems (MEDES 2013)*. ACM, 2013.

[18] Johannes Trame et al. *First Prototype of the Core Platform*. Public Deliverables, WP 2: Implementation Infrastructure. 2013.

[19] Christina Unger and Philipp Cimiano. 'Pythia: Compositional meaning construction for ontology-based question answering on the Semantic Web'. In: *Proceedings of the 16th international conference on Natural language processing and information systems* 6716 (2011), pp. 153–160.

[20] Ulli Waltinger, Dan Tecuci and Mihaela Olteanu. 'USI Answers: Natural Language Question Answering Over (Semi-) Structured Industry Data'. In: *Twenty-Fifth IAAI Conference* (2013), pp. 1471–1478.