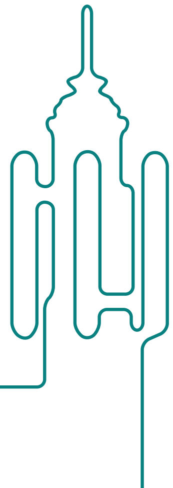
Thijs Vogels

SEARCHING THE SEMANTIC WEB

Academic Internship by Thijs Vogels
supervised by Dr. Hans de Bruin
in Spring 2014 at University College Roosevelt
and the HZ University of Applied Sciences



1 | Introduction

Hans de Bruin and his team at the HZ University of Applied Sciences have been working on methods to save and manage expertise and knowledge. In the context of the project ‘Zeeland Centraal’, they are currently building a knowledge database for water safety experts in The Netherlands. The unique feature of this knowledge database is that is contains rich semantic relations between concepts. In cooperation with water safety experts, an extensive thesaurus (SKOS) was built. The thesaurus tells which concepts are related to each other and in what way.

The thesaurus provides great opportunities in terms of searching the database. This document reports on an internship in which we contributed to the system by developing a method for using semantic information from this thesaurus to generate accurate and efficient search results. To reach this goal, we combine normal (full-text) search with the power of the semantic network. The thesaurus contains about 400 terms related to water safety and relations such as *broader*, *narrower*, *related* or *part of*.

Our search method is meant to be used for the water safety project and similar projects in other disciplines in the future. The method consists of three steps:

- Find matching documents for a given search string using full-text search using the open source software ElasticSearch.
- All documents are placed in one or more semantic contexts. From the full-text search results, we first determine the ‘search context’: a context that contains most the search results.
- All documents found are assumed to be pre-tagged with related thesaurus concepts by experts. By finding related concepts in the thesaurus, we add related documents to the search results. To find these related concepts, a spread-activation algorithm is used to spread the initial full-text search scores of the retrieved documents via the relations in the thesaurus.

The concept of the SKOS thesaurus and how it relates to the space of documents that should be searched in is illustrated in Fig. 1.1.

These pages provide an overview of the method as developed during the internship, but it also contains recommendations on how to extend on these methods in the future to get the most out of them. The document is meant as an internal reference manual for people who are working with these concepts.

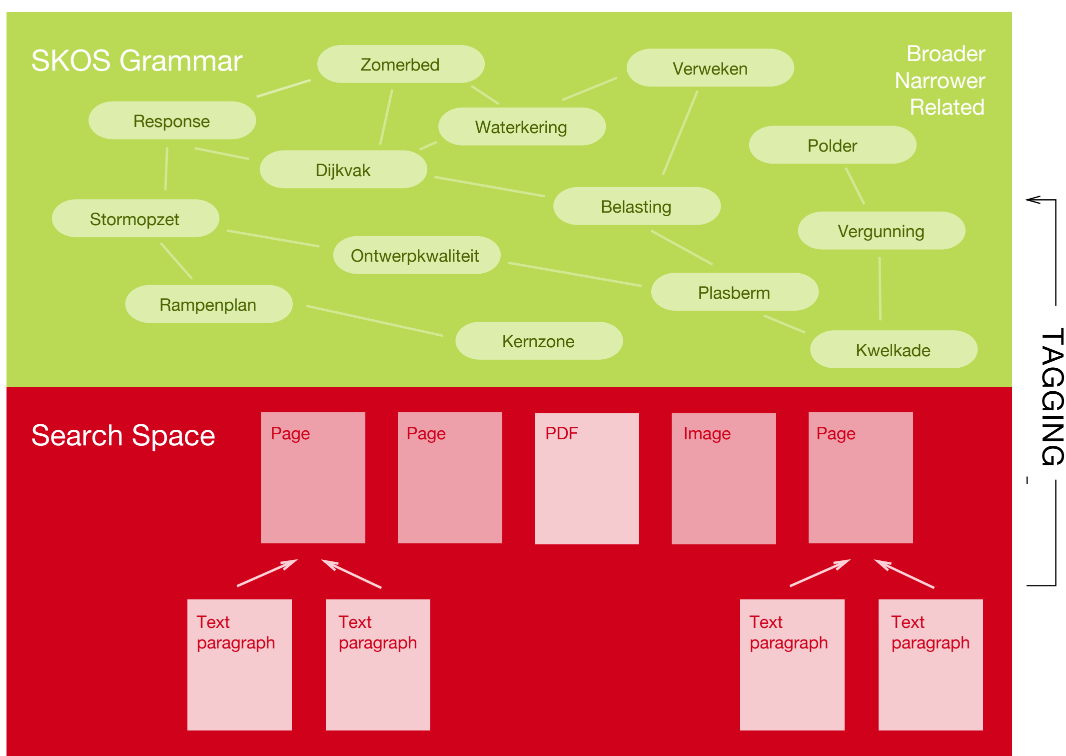


Fig 1.1: Schematic representation of the SKOS grammar. The red area (below) shows documents that are indexed by the system. These can be for example Wiki pages, images or PDF documents. These documents are all tagged with concepts from the SKOS Grammar (green area above), which indirectly relates all searchable content to each other.

Section 2 starts by giving an overview of the principles behind the 'Semantic Web'. After this, section 3 goes through all the tools that are used to implement our method and section 4 concludes with a step by step dissection of the implemented code.
Unfortunately, we have not been able to evaluate the precision and recall of our method yet, due to a lack of training data. This would be a desirable next step of this project.

Role of the Student My role as a student has been to conduct literature research, design and to finally implement the method. I implemented indexing and search code in PHP and JavaScript. As part of the internship, I presented my work at a meeting with *Bibliotheek.nl* in The Haguage. During the summer, we will likely continue this project to make sure all knowledge is transferred appropriately.

2 | The Semantic Web

The basis of this project lies in the principles of the 'Semantic Web'. Its philosophy is to create links between the wealth of information that is available on the world wide web [1]. Thousands of website have information about the same topics, but present these in a different format. Furthermore, they are usually written in natural language, so one cannot distinguish between Mars (the planet) and Mars (the company producing chocolate bars). The challenge is to define a set of rules that describe how web services should present data in such a way that they can easily be combined with the data from other sources.

In order to do this, it has to be perfectly clear what entities are described by a source. When different sources contain information about the same entity, it should be clear that they are talking about the same thing. To achieve this goal, entities are given a unique ID in the form of a Universal Resource Indicator (URI). When talking about a movie, for example, it would make sense to use its Internet Movie DataBase (IMDB) URL as an ID, because IMDB can be considered the authority in this field.

The relations between the entities too have unique ID's in the form of URI's. People can create their own relations, or use sets of well-defined and widely used ones. A well known example of a set of relations for describing inter personal relations is the Friend Of A Friend project and a standard for thesauri is SKOS.

All data is stored in *triples*: groups of three entities that together form a kind of 'sentence' such as (MichaelJackson born 1958) or (Head partOf Body), where the first entity is called the *subject*, the second the *predicate* and the last the *object*.

Relations are usually shared in the RDF document format, which is an XML format specified by the World Wide Web Consortium (W3C). It looks like this (please, do not spend hours trying to figure out what everything means at this point):

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <rdf:RDF
3   xmlns:dcterms="http://purl.org/dc/terms/"
4   xmlns:foaf="http://xmlns.com/foaf/0.1/"
5   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6   xmlns:schema="http://schema.org/">
7   <rdf:Description rdf:about="http://example.org/bob#me">
8     <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
9     <schema:birthDate
10      rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
11      1990-07-04
```

```
12     </schema:birthDate>
13     <foaf:knows rdf:resource="http://example.org/alice#me"/>
14     <foaf:topic_interest
15       rdf:resource="http://www.wikidata.org/entity/Q12418"
16     />
17   </rdf:Description>
18   <rdf:Description rdf:about="http://www.wikidata.org/entity/Q12418">
19     <dcterms:title>Mona Lisa</dcterms:title>
20     <dcterms:creator
21       rdf:resource="http://dbpedia.org/resource/Leonardo_da_Vinci"
22     />
23   </rdf:Description>
24 </rdf:RDF>
```

As a subset of XML, the RDF format can be parsed very quickly and is therefore suitable for machine use. It is, however, not very suitable for human eyes. A simpler syntax for the same documents is called *Turtle* and looks like this:

```
 1 @prefix dcterms: <http://purl.org/dc/terms/> .
 2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
 3 @prefix schema: <http://schema.org/> .
 4 @prefix dbpedia: <http://dbpedia.org/resource/> .
 5
 6 <http://www.wikidata.org/entity/Q12418>
 7   dcterms:title "Mona Lisa" ;
 8   dcterms:creator dbpedia:do_da_Vinci .
 9
10 <http://example.org/bob#me> a foaf:Person ;
11   schema:birthDate "1990-07-04" ;
12   foaf:knows <http://example.org/alice#me> ;
13   foaf:topic_interest <http://www.wikidata.org/entity/Q12418> .
```

The *Turtle*-file is much easier to read for mere humans. It starts by defining an number of prefixes: authorities in the form of a URL scheme that can be referenced to later on in the *Turtle*-file. The main content of a turtle file consists of subjects (printed in yellow), followed by relations that apply to these subjects. The example file above first defines what is meant by 'Mona Lisa' and that is was painted by Leonardo da Vinci. The relations 'creator' and 'title' are defined by dcterms. Next, a person called Bob is defined. Bob knows Alice and is interested in the Mona Lisa.

Simple Knowledge Organization System (SKOS)

Just like the relations *foaf:knows* and *dcterms:creator* we saw in the previous example, the Simple Knowledge Organization System (SKOS) defines a number of relations that are particularly useful for indicating relationships between concepts. The most important SKOS-relationships which we will be working in this projects are:

- skos:related
- skos:broader
- skos:narrower
- skos:partOf
- skos:instanceOf
- skos:subject

Related is used when two concepts are related to each other, but is not specified how. *Narrower* is used whenever one term is a 'subtype' of another relationship like in 'SteamShip skos:narrower WaterVehicle'. *Broader* is the inverse relationship. In other words: '*a* skos:narrower *b*' if and only if '*b* skos:broader *a*'.

PartOf should not be confused with *narrower*. *PartOf* only applies when a entity is a physical part of another entity, such as in 'IndexFinger skos:partOf Hand'. The use of *InstanceOf* is very straightforward: 'Noordzee skos:instanceOf Sea'.

The SKOS-relation *Subject* has a very specific usage. We will use it to tag all documents in the search space with concepts that they relate to. These concepts form the thesaurus and are connected via the relations defined by SKOS (see Fig. 1.1). If, for example the page 'Vergunningen Dijkbeheer' (Permits Dike Management) would be dealing with the concept 'Dijkvak' (dike section), the relation `:VergunningenDijkbeheer skos:subject :Dijkvak` would be created.

This section was meant to provide a basic understanding of the semantic relations that form the foundation of our search method. For more information on RDF, SKOS and the Semantic Web, please refer to [1].

3 | Tools

Before going into the actual search method, we will briefly describe the different tools that are used in this project. Like in the previous section, the descriptions are meant to provide an understanding of what the tools do and how they rougly operate. At the end of each section, one can find relevant links and tutorials if more in-depth knowledge is required.

Sparql

Sparql is a query language that is inspired by SQL. By its SQL-like syntax, it provides easy access to the semantic relations that are stored in an RDF-file (as shown in the last section), or in a *triple store* like Virtuoso or 4store. These triple stores can be seen as an equivalent of a traditional database. Also, the query language can be used to query many *Sparql Endpoints* on the web: as part of the 'semantic web'-philosophy, many web services provide open access to their semantic data using these endpoints. A number of available endpoints are listed on http://www.w3.org/wiki/SparqlEndpoints.

In order to illustrate some basic Sparql queries, we go through some examples as presented in [1]. Imagine having access to a triple store that has information about movies and actors. The dataset contains triples as `:JonnyDepp :playedIn :PiratesOfTheCaribbean`, as well as `:PiratesOfTheCaribbean :hasDirector :GoreVerbinski`. We can now write a Sparql query to retrieve a list of all movies and their directors that are in the database:

```
1 SELECT ?movie, ?director
2 WHERE{ ?movie :hasDirector ?director . }
```

This query illustrates the default format of a Sparql query: it *selects* certain features from triples that match a pattern (*where*). This pattern can be simple and just describe one triple with wild-cards (indicated by a question mark) like in the previous example, but it can also take a more complex form. Say, for instance, that we want a list of all the actors that played in any film that was directed by Gore Verbinski. We would then use the following query:

```
1 SELECT ?actor
2 WHERE{ ?movie :hasDirector :GoreVerbinski .
3        ?actor :playedIn ?movie . }
```

In this case, the first line of the pattern is matched first. Intuitively, Sparql finds all movies that are directed by Gore Verbinski and secondly, the matched actors are fed into the second pattern that in turn selects the corresponding actors.

This short introduction does no justice to the broad capabilities of Sparql. The language also includes things like sorting, filters, negation, transitive queries (really cool) and much more. For a more detailed tutorial into Sparql, please refer to [1].

In the beginning of this project, the Sparql language played an important role in this internship. We planned to use it as a basis for the semantic search method, but changed our minds. The main reason for this was immaturity of the interface between Semantic Media Wiki—the system in which all information is stored—and the triple store that we use, Virtuoso.

Semantic MediaWiki

The search methods that were developed during the internship are to be implemented on various Wikis developed by Hogeschool Zeeland, the first of which being about water safety. These Wikis are built on the foundation of *MediaWiki*, the software that also powers Wikipedia. We use the MediaWiki software with an additional extension that enables the persistence of rich semantic information: *Semantic MediaWiki*.

In the philosophy of MediaWiki, all entities in the system have their own page. In our case, the entities in the Wiki are for example:

- **SKOS Concepts:** A list of (now around 250) concepts that form a domain specific thesaurus for the Wiki. For the water safety Wiki, these are for example 'dijkvak' (dike section), 'water' and 'waterhoogte' (water level). Semantic relations between the concepts (related, broader, narrower, partOf) were added by water safety professionals and are stored in the semantic Wiki.
- **Information pages:** Pages about all kinds of procedures for checking dikes, about emergy situations and how to evacuate in these situations and much more.
- **Files:** Files and links that are referred to in a Wiki page also get their own 'page'. This page contains some information about the file, the content-type and other meta data.
- **Instances of SKOS Concepts:** These can be seen as a subtype of the information pages. For a SKOS concept such as 'dike section', the eleven physical dike sections in the province of Zeeland have a page of their own that includes factual information and also links to experiences related to that particular dike section.

Note that this list is not exhaustive, but is meant to go through the pages that are most relevant for searching.

Semantic MediaWiki enables semantic relations between these pages to be captured. For instance, the fact that ‘Borrendamme’ is an instance of a dike section is stored as a triple like `:Borrendamme rdf:instanceOf :DikeSection`. Other important relations are ‘depends’, ‘decomposes’ and ‘contributes’. Also, most of the factual data of pages is stored in a set of pre-defined relations. ‘Borrendamme’ is in the area of ‘Schouwen-Duiveland’, so we have `:Borrendamme :location :SchouwenDuiveland`. In these examples, the ID’s `:Borrendamme`, `:DikeSecton` and `:SchouwenDuiveland` are shorthands for the URL’s of their Wiki pages.

Another relevant feature of Semantic MediaWiki is that it ensures every entity in the system to be placed in one or more ‘contexts’ that are pages of their own. Even the contexts themselves have a super context, such that contexts form a tree. Contexts are often situations; scenarios for information. ‘Evacuating Ameland’, for instance, could be a context that holds information on emergency measures for a possible flood in the island of Ameland. The contexts play a key role in our search method. We assume that our contexts have great overlap with the ‘situations’ in which a user of the Wiki could be searching for information.

Pages in MediaWiki are created by users through pre-defined forms. Depending on the type of page, certain factual information is asked for. The resulting page is stored in a text format called *WikiText*. As an example, below is the source for the page of ‘Borrendamme’:

```
1 {{Dijkvak
2   |Description=Het dijktraject Borrendamme Polder Schouwen,
3    Cauwersinlaag, Havenkanaal West ligt aan de zuidzijde van
4    Schouwen- Duiveland aan de Oosterschelde.
5   |Supercontext=Projectbureau Zeeweringen,
6   |Locatie=Schouwen-Duiveland
7   |Coordinaat=51.640074,3.890398
8   |Begindijkpaal=dp 161+10m
9   |Einddijkpaal=dp 198+50m
10  |Steenbekleding=Betonzuil
11  |Voorland=Stroomgeul
12 }}
13 {{Dijkvak werkproces creatie}}
14 {{Paragraphs show}}
15 {{Kerndocumenten show}}
16 {{Beeldmateriaal show}}
17 {{Heading
18   |Heading nl=Borrendamme
19 }}
20 {{Paragraph
```

```
21   |Paragraph number=1
22   |Paragraph language=Dutch
23   |Paragraph=Het dijktraject Borrendamme Polder Schouwen,
24    Cauwersinlaag, Havenkanaal West ligt aan de zuidzijde van
25    Schouwen- Duiveland aan de Oosterschelde. Het traject
26    ligt tussen dp 160 en dp 207 (keersluis bij Zierikzee).
27    De totale lengte bedraagt ongeveer 4,6km, het dijkvak valt
28    onder het beheer van waterschap Scheldestromen. Voor een
29    deel van het traject ligt de geul Roompot, welke ter
30    hoogte van het Westelijk Havenhoofd plaatselijk een diepte
31    heeft van meer dan 50m. Het andere deel van het traject
32    betreft de westelijke zijde van het Havenkanaal van
33    Zierikzee. Ter plaatse van de havenmonding bevindt zich het
34    Westelijk Havenhoofd.
35 }}
36 {{Kerndocument
37   |Kerndocument=Media:Planbeschrijving Borrendamme, Polder
38    Schouden, Cauwersinlaag, Havenkanaal West 075597494 VB.pdf
39   |Kerndocument type=Projectplan
40 }}
41 {{Kerndocument
42   |Kerndocument=Media:Passende_beoordeling_Borrendamme.pdf
43   |Kerndocument type=Vergunningen en ontheffingen
44 }}
45 {{Kerndocument
46   |Kerndocument=Media:2012011791_definitieve_nb-wetvergunning_
47    Borrendamme,_Cauwersinlaag,_Havenkanaal_West.pdf
48   |Kerndocument type=Vergunningen en ontheffingen
49 }}
50 {{Kerndocument
51   |Kerndocument=Media:D2 Ontwerpnota Borrendamme getekend.pdf
52   |Kerndocument type=Ontwerpnota
53 }}
54 {{Kerndocument
55   |Kerndocument=Media:Samenvatting planbeschrijving Borrendamme,
56   Polder Schouwen, Cauwersinlaag, Havenkanaal West 07561.pdf
57   |Kerndocument type=Projectplan
58 }}
59 {{Context query}}
```

Let's interpret this document. It consists of a number of templates that are called by double curly brackets. For 'Borrendamme', the first template that is called is 'Dijkvak' (dike section). The template has a page of its own (Template:Dijkvak) and is called with a set of parameters: description, supercontext, etc. The template makes sure that the provided parameters are mapped onto the correct semantic relationships internally. The particular template for

`Dijkvak` also links the page to some thesaurus concepts by using another template: 'Resource Description'. This becomes clear from inspecting part of the source of `Template:Dijkvak`.

```
1 {{Resource Description
2   |subject=Dijkvak, Dijk, Waterkering
3 }}
```

Looking back to the source of 'Borrendamme', we see that it includes many more templates that were all created by filling in the Dijkvak form. It includes paragraphs, a little bit of view information, and also generates a view of related pages at the bottom of the page.

Forms and templates are a flexible way to create complex semantic structures that can easily be managed by users that are not computer science experts. This is considered to be the main advantage for using Semantic MediaWiki in HZ's projects: the Wiki environment takes care of bridging abstract concepts to the end users through a clear interface. Another great advantage of the system is that it is a nice environment to 'play around'. When starting a project, one does not have to directly implement a final version, but rather, he or she can change bits and pieces of the code during the course of the project, as new ideas develop.

The flexibility of the Wiki-system certainly also has its disadvantages. By storing pages in the kind of text format as presented above, optimization becomes a non-trivial task. One look at a Wiki-database is enough to confirm this. It would certainly be an option to, once the principles that are developed in this projects mature by intensive usage, transfer the system to a less flexible, dedicated environment.

One more thing that is of interest to our project is that Semantic MediaWiki offers the possibility to synchronize its data store with triple stores. The store of choice for this project is Virtuoso, an open source triple store that is maintained by *OpenLink*. Unfortunately, the link between the two components is not yet stable enough to be used in production. Therefore, we choose to resort to *ASK Queries* instead of *Sparql* queries on Virtuoso for programmatic data access at the moment.

To learn more about (Semantic) MediaWiki, have a look at their websites: http://www.mediawiki.org/ and http://semantic-mediawiki.org/.

ASK Queries

Semantic MediaWiki features ASK, a simple query language for semantic searching. It is nowhere near as rich and extensive as Sparql and not optimized for performance, but it is easy to learn and suffices in most simple use cases. ASK Queries can be called from any page

in the Wiki to show dynamic info graphics and tables. Let's look at a simple example of an ASK query from the Semantic MediaWiki website. The query could be included in the source of a page.

```
1 {{#ask: [[Category:City]] [[located in::Germany]]
2 | ?population
3 | ?area#km² = Size in km²
4 }}
```

The query will return a table with two columns: 'population' and 'area' and as rows the cities in Germany. Dissecting the query, we first encounter `{{#ask`. This is the standard syntax for calling a function in the WikiText language. Secondly, we see two terms that are enclosed by double square brackets. The first, `Category:City`, selects all pages that inherit from this category, and the second, `located in::Germany`, narrows down the selection by only keeping pages that have the `?page :locatedIn :Germany` relation. Mind the difference between the single and the double colons in these queries. The single colon is part of the ID of the category, whereas the double colon indicates the object of a semantic relation. The second part of the ASK query contains the projection of properties that will be shown in the resulting table, in this case 'population' and 'area in square kilometers'.

Similarly, we could request all the cities with an area that is larger than 100 squares kilometers like this:

```
1 {{#ask: [[Category:City]] [[area::>100km]]
2 | ?population
3 | ?area#km2 = Size in km2
4 | ?located in
5 }}
```

What if we wanted to obtain a list of all actors that were born in a city in Germany? This is another example from the Semantic MediaWiki website. We can accomplish this by using so called sub queries:

```
1 {{#ask: [[Category:Actor]] [[born in::
2     <q>[[Category:City]] [[located in::Germany]]</q>
3   ]]
4 | ?name
5 }}
```

The sub query in this example is indicated by `<q></q>`. A comprehensive overview of all selection options for ASK queries can be found on http://semantic-mediawiki.org/wiki/Help:Selecting_pages.

Apart from being able to incorporate ASK queries into Wiki pages, Semantic MediaWiki also includes an external HTTP API that allows for ASK queries from outside. This API is accessible via `http://hostname/api.php?action=ask&query=...`. The API is also accessible internally from within the PHP source of the Wiki. For Wiki-plugins like our search page, this method is to be preferred over the HTTP API because it is considerably faster.

Although is it not immediately clear from the basic examples of the two semantic query languages ASK and Sparql in this paper, Sparql is definitely a lot more powerful and performant than ASK. It would therefore be desirable in the future to use Sparql as an interface to the semantic data from a Wiki in stead of ASK.

ElasticSearch

ElasticSearch is a free enterprise class search engine that is build on the famous open source project Lucene. It takes care of creating search indices, possibly spreading the indices over multiple machines in a cluster and the actual searching it self.

ElasticSearch is easy to set up. Simply download the installation files from their website and execute `bin/elasticsearch` to start up a simple HTTP server, by default at `localhost:9200`. This server serves as the one and only interface to ElasticSearch. Making indices, updating them and the actual querying are all done through simple HTTP calls.

Indices are subdivided into 'types', each of which may contain arbitrary documents with their data stored as a JSON document, a simple text format for key/value objects. Furthermore, each of these documents are supposed to have a unique ID. Suppose that our index contains a type 'movie'. The document at `localhost:9200/hzbwnature/movie/1` (where 1 is the document's ID) could look like this:

```
1 {
2   "title": "Pirates of the Caribbean: The Curse of the Black Pearl",
3   "director": "Gore Verbinski",
4   "year": 2003
5 }
```

In theory, movies are not required to all have the same three attributes. Other movies could have more or less data in the index. A new movie can be added to an index through the HTTP API:

```
1 curl -XPUT "http://localhost:9200/hzbwnature/movie/2" -d '
2 {
3   "title": "Pirates of the Caribbean: Dead Man s Chest",
4   "director": "Gore Verbinski",
5   "year": 2006
6 }'
```

Searching too goes via HTTP. Let's first get a list of all movies in our index:

```
1 curl -XGET "http://localhost:9200/hzbwnature/movie/_search" -d '
2 {
3   "query": {
4     "match_all": {}
5   }
6 }'
```

Well, that was easy. We can also do a basic free text search through the whole index for movies that contain the word 'ship', like this:

```
1 curl -XPOST "http://localhost:9200/hzbwnature/_search" -d '
2 {
3     "query": {
4         "query_string": {
5             "query": "ship"
6         }
7     }
8 }'
```

By default, ElasticSearch uses all indexed data to search in, but we can also indicate the fields:

```
1 curl -XPOST "http://localhost:9200/hzbwnature/_search" -d '
2 {
3     "query": {
4         "query_string": {
5             "query": "ship",
6             "fields": ["title"]
7         }
8     }
9 }'
```

ElasticSearch queries return a JSON object containing the results and attached to them a numeric score for relevance: The movie "Ship" would for example get a higher score

than a movie of which the description field incidentally includes that word once in a 200-word piece.

`query_string` is probably the simplest query that ElasticSearch offers. It does, however, also allow for more complicated queries, such as `fuzzy`, `match_all`, `regexp`, `terms`, `wildcard`, `fuzzy_like_this`, `span_near` and many more. A complete list is readily available on ElasticSearch's website.

Next to a multitude of query operators, ElasticSearch also offers a range of filters. These can be thought of as queries, with as difference that they produce a hard cut-off point instead of scored documents. Documents either match a filter or not. This makes filters perform faster than queries.

There is one other relevant area of ElasticSearch that needs a bit of explanation. ElasticSearch offers a set of 'analyzers' that are used to prepare queries and data before searching in them. Analyzers could for example remove stopwords, or stem the words to make sure that (almost) equivalent words like 'pirate' and 'pirates' match exactly. Again, a list of available analyzers is available on ElasticSearch's website. In addition to the standard library of analyzers, we make use of an open source project by Jörg Prante: elasticsearch-analysis-skos . His analyzer can read an RDF file and expand queries by adding equivalent terms (through the `:altLabel` relationship.)

In addition to ElasticSearch's HTTP API, there are wrappers available for many languages. In our project, we use PHP and Javascript wrappers to take care of error handling and such.

4 | Search method

Now that we are equipped with sufficient knowledge about the tools that make our search method possible, this section will go into depth about the actual search method. The section first introduces the ontology that forms the semantic structure of the Wiki, after which the full method is outlined and explained part by part. The later part will be heavy on implementation code, since this is assumed to be helpful for someone who would later pick up this work to continue on it.

Literature research has shown that ontology-based searching is definitely not a trivial problem [6]. Most methods only provide good results for very specific information bases and queries and performance is often difficult to compare. Many researchers obtain good recall by using query expansion techniques [5] [3]. This means that user queries are enriched by adding related terms to the queries through an ontology. Mostly, the ontologies in question are computer generated based on statistical models [4]. Akrivas et al. [3] go a step further than simple term expansion. Instead of expanding terms, they identify semantic concepts that can consist of multiple terms and they expand on these. Furthermore, by identifying a 'query context', they manage to narrow down the results to greatly enhance precision.

Rocha et al. [7] measure particularly good results with a hybrid approach to semantic searching. They combine spread activation techniques and query expansion with regular full-text search and a symbolic approach to achieve better results when compared to each of the methods alone.

In our method, we combine the hybrid spread activation techniques as presented by Rocha et al. with the notion of a 'query context', like in the work of Akrivas et al. Another distinctive feature is that we can work with an ontology that is designed and curated by a team of professionals, rather than automatically generated. We believe that this will bring a boost in our precision. This is only possible since the Wiki's we work with are extemely specialized. Curated semantic relations do have the implication that our thesaurus is relatively sparse compared to the automatically generated examples. This is something that should be taken into account throughout our algorithms.

The Ontology and the Semantic Network

The innovative power of this project lies in the structure of its ontology. It is convenient to explain the semantic structure of our product by going through the cells of Fig. 4.1.

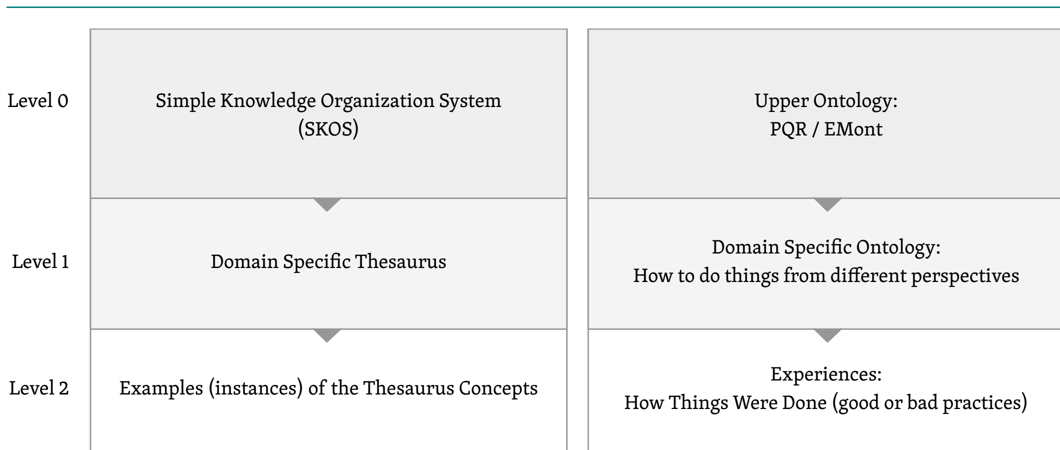


Fig 4.1: Levels in the Semantic Network. The left branch shows factual information, while the right branch contains human knowledge, experiences and good or bad practices. Vertically, level 0 contains a high level abstract 'upper ontology', level 1 contains a domain specific interpretation of the concepts defined in level 0 and level 2 includes practical examples. Between a lower level and the level above, one can always read the 'instance of'-relation.

On the one hand (the left branch of the chart), we store factual data. We store semantic relations between these facts using the ontology of the 'Simple Knowledge Organization System' (SKOS). SKOS prescribes a set of relations that could exist between concepts and it describes the properties of these relations. SKOS forms 'level 0' of this side of the scheme.

For each specific project that uses our system, a domain specific thesaurus is constructed. It includes a specialized set of concepts and the relations between them (related, broader, narrower, part of, ...) For each of the concepts in the thesaurus, we can read `:instanceOf skos:Concept`. Going down to level 2 of this branch, we find instances of the concepts in level 1. If a concept in level 1 is 'dike section', Borrendamme would be an instance of this in level 2.

Now we look at the other side of the chart. The upper ontology that describes possible relations and entities is EMont, an ontology developed by Hans de Bruin at HZ. The ontology is inspired by the Soft Systems Methodology [8] and the idea is very intuitive. EMont recognizes issues or goals, denoted by the letter R. In order to achieve such a goal R, several activities, indicated by P can be done. The relation between a P and R is always 'contributes to'. The actual steps taken to do an activity P, can be called Q. The relation between a P and a Q is 'part of'. EMont allows P's and Q's to be nested: activities to depend on each other and it allows conditions to be defined on the concepts.

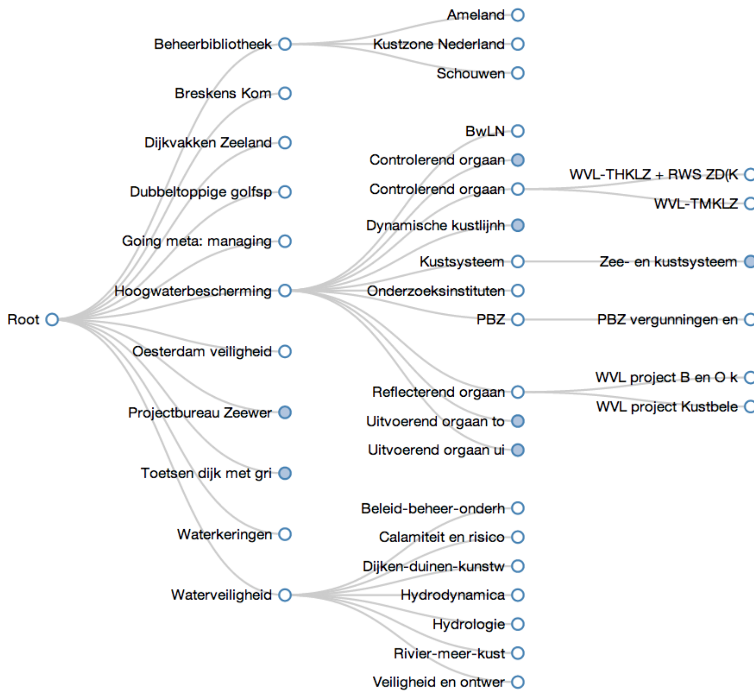


Fig 4.2: An excerpt from the context hierarchy of the HZ Building With Nature Wiki (Dutch).

Like on the left side, the upper part of the branch contains EMont: the bare definition of what a P, Q and R mean and what relations exist between them. On the next level, we get a domain specific interpretation of the ontology. Level 1 contains actual problems and activities that are related to the domain of a Wiki and level 2 features examples and experiences of doing the activities in level 1 that have been collected over the years.

The six sections of Fig. 4.1 do not live a life of their own. Activities and issues on the right side are linked to specific skos concepts (tagged) by the 'subject' relation. For our search method to work out, we make the assumption that all pages are properly tagged with related SKOS concepts. This is not yet the case for our first test-wiki, 'HZ Building with Nature', but the tagging is in full swing at the moment.

Like mentioned before, we also assume the entities in our Wiki to be placed in one or more 'contexts', which we use to ensure good precision in search results. Contexts are often related to the R's of the PQR formula (issues and situations). This makes sense, because a user is often searching for information on how to solve a particular issue. Contexts are hierarchical. Fig. 4.2 shows an excerpt of the context hierarchy.

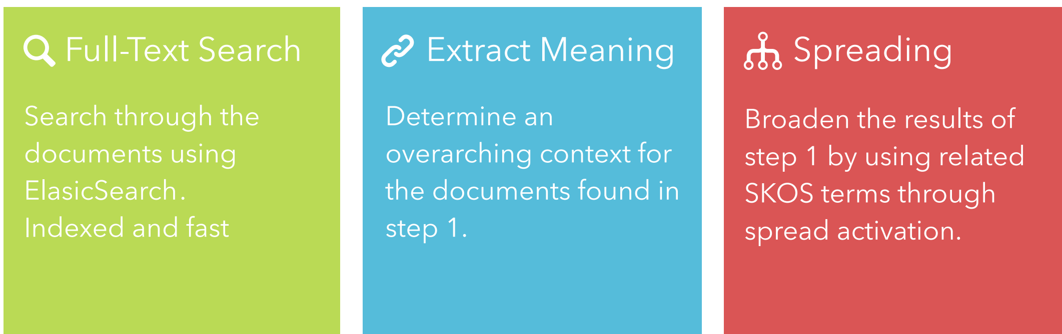


Fig 4.3: The three steps of the search method.

Outline

The search method that was developed during this internship consists of three steps, as illustrated in Fig. 4.3. The first step is to do a normal full-text search on the document titles, their content, context and other relevant attributes. These attributes are stored in an ElasticSearch index that allows this step to be very performant. The result of this step is a list of documents with an assigned score for relevance. In the second step, we find an overarching 'search context' that captures the most of the found documents of step 1. The search context is meant to be the essence of the search query and is expected to be key to the success of our method. Lastly, we use the thesaurus concepts that all documents in the Wiki are tagged with to broaden our results by spread activation. We use the initial scores of the pages to give weight to the tags of the pages, and through this, find more pages that could be relevant to the user. This will increase recall. To also ensure good precision, we restrict the spreading to the bounds of the determined search context.

In the coming sections, we will go in detail through the implementations of the methods, where present.

Indexing

This section goes through the process of creating the ElasticSearch index that is a prerequisite for step 1 of Fig. 4.3. The indexing code was written in PHP and it is to be run at fixed times (every hour or every day, for instance). In the future, we could adapt the indexing to update the index real-time when a document changes.

We build the index `hzbwnature` that has three types: SKOS concepts, contexts and 'intentional elements'. The last type is basically a category of all Wiki pages that are meaningful to index. In the future, another index for files (pdf, image) that should be indexed is desirable.

The code uses *Composer* for dependency management in PHP. We use ElasticSearch's PHP wrapper instead of direct HTTP calls to the ElasticSearch server (still running at `localhost:9200`. The wrapper makes sure that errors are handled properly.) The full code repository for the project is available at GitHub.

We start by some setup code:

```
1 # Include a class that allows for sending ASK queries to the wiki
2 include_once('classes/AskApi.php');
3 # Load Composer's autoloader for the ElasticSearch class
4 include_once('vendor/autoload.php');
5
6 # Settings
7 $wiki = "wiki.local"; // my local copy of the wiki
8 $elasticIndexDir = "/tmp/elasticindex";
9 # Next is a file with the domain specific skos in N3 format
10 $skosN3file = "...path.../hzbwnature.n3";
```

Next, we instantiate clients for ElasticSearch and ASK:

```
1 # create an associative array with parameters that tell ES what
2 # what to delete
3 $deleteParams['index'] = 'hzbwnature';
4 # delete:
5 try {
6   $elastic->indices()->delete($deleteParams);
7 }
8 catch (Exception $e) {
9   // handle exceptions properly
10 }
```

This step should in fact not be ran every time the index is refreshed, but it is important to be able to remove and reset the whole index. If the index is removed, we need to configure it again:

```
1 # Construct an associative array with properties
2 $indexParams['index']  = 'hzbwnature';
3
4 # Specify the usage of the SKOS equivalence filter, just like simple
```

```
5 # dutch stemming and stop word removal
6 $indexParams['body']['settings']['index']['analysis'] = array (
7   "filter" => array (
8     "skosfilter" => array (
9       "type" => "skos",
10      "path" => $elasticIndexDir,
11      "skosFile" => $skosN3file,
12      "expansionType" => "URI"
13    )
14  ),
15  "analyzer" => array (
16    "skos" => array (
17      "type" => "custom",
18      "tokenizer" => "keyword",
19      "filter" => "skosfilter"
20    ),
21    "my_analyzer" => array (
22      "type" => "snowball",
23      "language" => "Dutch",
24      "stopwords" => array("aan","af","al",...,"zou")
25    )
26  )
27 );
28
29 # Indicate on which fields the above defined analyzers should work
30 # Analyzers can be applied during searching, during indexing, or both.
31 $indexParams['body']['mappings']['_default_']['properties']
32  ['subject'] = array (
33    "type" => "string",
34    "index_analyzer" => "skos",
35    "search_analyzer" => "standard"
36  );
37 $indexParams['body']['mappings']['_default_']['properties']
38  ['content'] = array (
39    "type" => "string",
40    "index_analyzer" => "my_analyzer",
41    "search_analyzer" => "my_analyzer"
42  );
43 $indexParams['body']['mappings']['_default_']['properties']
44  ['title'] = array (
45    "type" => "string",
46    "index_analyzer" => "my_analyzer",
47    "search_analyzer" => "my_analyzer"
48  );
49
50 # Create the index (still needs appropriate error handling)
```

```
51 $elastic->indices()->create($indexParams);
```

Now, let's look at the code that indexes all SKOS concepts. First, we obtain a list of SKOS Concepts using an ASK call:

```
1 $concepts = $ask->query('
2   [[Category:SKOS Concept]]
3   |?skos:altLabel
4   |?skos:related
5   |?skosem:narrower
6   |?skosem:broader
7   |?skosem:partOf
8   |?skos:definition
9 ');
```

and then we loop through them to add them to the index. As a unique ID for each document in the index, we use the MD5 hash of its URL.

```
 1 foreach ($concepts as $c) {
 2   $params = array();
 3   $params['body'] = array(
 4     "url" => $c->fullurl,
 5     "skos:prefLabel" => $c->fulltext,
 6     "skos:altLabel" =>$c->printouts->{'Skos:altLabel'},
 7     "skos:definition" =>$c->printouts->{'Skos:definition'},
 8     "skos:related" =>
 9       array_map(function ($a) {
10         return $a->fullurl;
11       }, $c->printouts->{'Skos:related'}),
12     "skos:narrower" =>
13       array_map(function ($a) {
14         return $a->fullurl;
15       }, $c->printouts->{'Skosem:narrower'}),
16     "skos:broader" =>
17       array_map(function ($a) {
18         return $a->fullurl;
19       }, $c->printouts->{'Skosem:broader'}),
20     "skos:partOf" =>
21       array_map(function ($a) {
22         return $a->fullurl;
23       }, $c->printouts->{'Skosem:partOf'})
24   );
25   $params['index'] = 'hzbwnature';
26   $params['type'] = 'skos_concept';
```

```
27   $params['id'] = md5($c->fullurl);
28   $returnValue = $elastic->index($params);
29 }
```

Now, we will make an index of all available contexts.

```
 1 # Retrieve the contexts and the hierarchy via ASK
 2 $contexts = $ask->query('
 3   [[Category:Context]]
 4   |?Category
 5   |?Supercontext
 6 ');
 7
 8 # Loop through the contexts and add them to the index
 9 foreach ($contexts as $c) {
10
11   $params = array();
12
13   # Determine the supercontext, if there is none given, it is ROOT
14   $super = 'ROOT';
15   if (count($c->printouts->{'Supercontext'}) > 0)
16     $super = $c->printouts->{'Supercontext'}[0]->fullurl;
17
18   # Set the index content
19   $params['body'] = array(
20     'url' => $c->fullurl,
21     'name' => $c->fulltext,
22     'supercontext' => $super,
23     'category' =>
24       array_map(function ($a) {
25         return $a->fullurl;
26       }, $c->printouts->{'Category'}),
27     'category_readable' =>
28       array_map(function ($a) {
29         return $a->fulltext;
30       }, $c->printouts->{'Category'})
31   );
32
33   $params['index'] = 'hzbwnature';
34   $params['type'] = 'context';
35   $params['id'] = md5($c->fullurl);
36   $returnValue = $elastic->index($params);
37   # again, proper error checking is desirable
38 }
```

Before we can index the documents, we should realize that the contents of the documents (paragraphs) are stored in separate sub-objects that can not so easily be queried. Therefore, we first ASK for all paragraphs and their page and construct a dictionary that contains lists of all paragraphs per context.

```
1 $paragraphs = $ask->query('
2 [[Paragraph::+]]
3 |?Paragraph
4 |?Paragraph subheading
5 |?Paragraph language
6 |?Paragraph number
7 |?Paragraph back link');
8
9 # Create an empty dictionary for the paragraphs of an element
10 $elementPars = array();
11 # Loop through the paragraphs and add them to the dictionary
12 foreach ($paragraphs as $p) {
13   $url = $p->printouts->{'Paragraph back link'}[0]->fullurl;
14   $elementPars[$url][] = $p;
15 }
16 # Define a function that returns the paragraphs belonging to a
17 # URL.
18 function getParagraphs($url) use ($elementPars) {
19   if (isset($elementPars[$url])) return $elementPars[$url];
20   else return array();
21 }
```

With the paragraphs, we can continue to index the actual intentional element. This is the last part of the indexing that is currently active:

```
1 # Retrieve the elements
2 $elements = $ask->query('
3   [[Category:Intentional Element]]
4   [[Context::+]]
5   |?Concerns
6   |?Context
7   |?Category
8 ');
9
10 # Loop through them and add them to the index
11 foreach ($elements as $element) {
12   # Retrieve the paragraphs of this page from the dictionary
13   $pars = getParagraphs($element->fullurl, $elementPars);
14
```

```
15   # Add to the index
16   $params = array();
17
18   # Combine the paragraphs into one concatenated content string
19   $content = implode(
20     array_map(
21       function ($p) {
22         return implode($p->printouts->{'Paragraph subheading'}," ") .
23                "\n " .
24                implode($p->printouts->{'Paragraph'}," ");
25       },
26       $pars
27     )," \n");
28
29   # Set the parameters and index it
30   $params['body'] = array(
31     "url" => $element->fullurl,
32     "title" => $element->fulltext,
33     "content" => $content,
34     "concerns_readable" => implode(
35       array_map(function ($a) {
36         return $a->fulltext;
37       }, $element->printouts->{'Concerns'}), " "),
38     "concerns" =>
39       array_map(function ($a) {
40         return $a->fullurl;
41       }, $element->printouts->{'Concerns'}),
42     "context_readable"=> implode(
43       array_map(function ($a) {
44         return $a->fulltext;
45       }, $element->printouts->{'Context'}), " "),
46     "context"=>
47       array_map(function ($a) {
48         return $a->fullurl;
49       }, $element->printouts->{'Context'})
50   );
51   $params['index'] = 'hzbwnature';
52   $params['type'] = 'intentional_element';
53   $params['id'] = md5($element->fullurl);
54   $returnValue = $elastic->index($params);
55 }
```

Searching: Query and Result expansion

This section deals with the actual searching. We show the implementation of steps 1 and 2 from Fig. 4.3. Step 3, the spread activation, could not be implemented yet due to a lack of

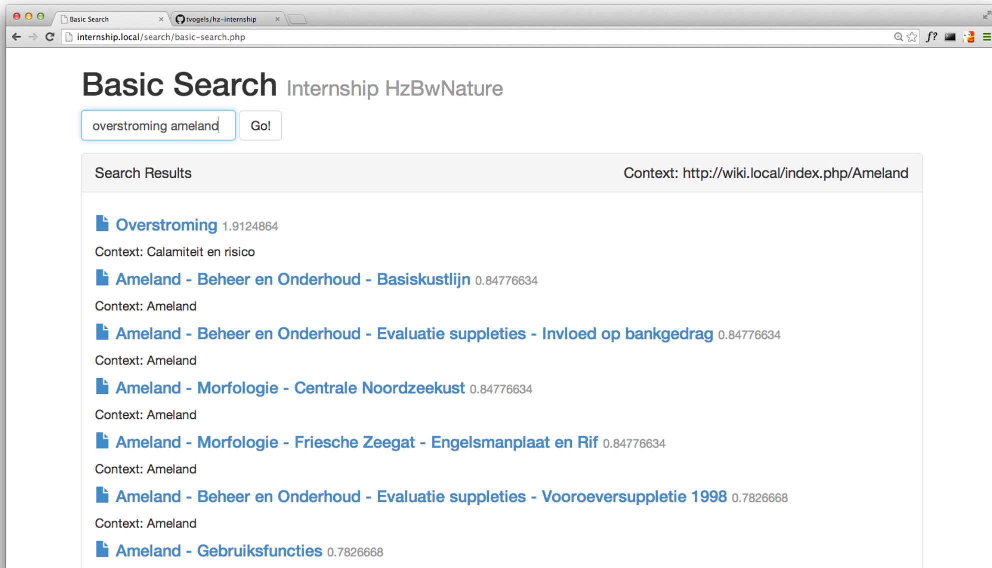


Fig 4.4: Screenshot of the test environment for searching. The documents returned by ElasticSearch for the query string 'overstroming Ameland' are shown with their corresponding scores (in light gray).

tagged test documents. We will explore this further during the summer, but for now, please refer to [7] for more details on the last step.

We will first show how to retrieve search documents from ElasticSearch, and then we go through the algorithm that determines the overarching context of the search results. The code from now on was implemented in JavaScript, to maximize responsiveness of the results.

We start by initializing the application and

```
1 // Declare some variables
2 var $field, $form, $results, client, p;
3
4 // Create an ElasticSearch client for our server
5 client = new $.es.Client({
6   'host': 'localhost:9200'
7 });
8
9 // Refer to certain areas of the web page that need to be controlled
```

```
10 // in this script.
11 $field = $('#search-query');
12 $form = $('#search-form');
13 $results = $('#results');
14
15 // Set the status of the page to 'loading':
16 $results.addClass('csspinner');
17
18 // Retrieve all contexts from the index.
19 // The request is asynchronous and returns a promise.
20 // We can add listeners to the promise that will be called
21 // when it fails or succeeds. This is done in a later
22 // code block.
23 contextsPromise = client.search({
24   index: 'hzbwnature',
25   type: 'context',
26   size: 1000,
27   body: {
28     query: {
29       match_all: {}
30     }
31   }
32 });
```

When the promise for a list of contexts resolves, we will setup the functions that are used to determine the overarching context of search results. The algorithm works as follows:

- For each context, keep track of their 'weight'.
- For each document that was found, add its score to the weight of its context.
- When weight is added to a context, the same weight is added to the super context. Since the contexts form a tree, this algorithm converges. Furthermore, the ROOT node, will always contain the total weight.

In order to determine the overarching node of the search results, we will now start at the root of the tree and identify the deepest node that has at least 80% of the total weight of the system as the 'overarching node' and 'search context'.

The next of block will be called when the `contextsPromise` resolves. It prepares the contexts for our algorithm and defines helpers functions for adding weight to the context-tree nodes.

```
1 contextsPromise.then(function(e) {
```

```
2
3   // Declare a bunch of local variables
4   var a, addWeight, children, findContext, result_template, source,
5       supercontext, _i, _len, _len2, _ref;
6
7   // The data is loaded now, so remove the spinner
8   $results.removeClass('csspinner');
9
10  // Construct a dictionary of context -> supercontext pairs
11  supercontext = {};
12  _ref = e.hits.hits;
13  for (_i = 0, _len = _ref.length; _i < _len; _i++) {
14    a = _ref[_i];
15    supercontext[a._source.url] = a._source.supercontext;
16  }
17
18  // Construct a similar dictionary for the other direction.
19  // The dictionary will return a list of children for a given node
20  children = {};
21  for (_i = 0, _len2 = ref.length; _i < _len2; _i++) {
22    a = ref[_i];
23    children[a._source.supercontext].push(a._source.url);
24  }
25
26  // Define a function that recursively adds weight to a node
27  // and all its root nodes. We do not protect against infinite
28  // looping yet.
29  // Its input values are
30  // (1) the url of the node
31  // (2) the weight to add to the node
32  // (3) a reference to the dictionary in which the weights are
33  //     stored. The storage dictionary is formatted as [url: weight]
34  addWeight = function(url, weight, storage) {
35    var sc;
36    if (storage[url] == null) {
37      // If there is no entry in the storage dictionary for this URL,
38      // create an entry
39      storage[url] = weight;
40    } else {
41      // Else, add the weight to the current weight of the node
42      storage[url] = storage[url] + weight;
43    }
44
45    // retrieve the supercontext
46    sc = supercontext[url];
47
```

```
48      // As long as we are not the top of the tree, add weight to
49      // the supercontext
50      if (url !== "ROOT") {
51        storage = addWeight(sc, weight, storage);
52      }
53      return storage;
54    };
55
56    // Now define a function that finds the overarching context
57    // starting at the root, it goes down until there is no single
58    // branch that has a weight less than 'minimum'
59    findContext = function(node, weights, minimum) {
60      // Declare some variables
61      var c, childWeights, maxWeight, weight;
62
63      // A small helper function that return the weight of a context
64      // in the storage dictionary 'weights' if there is an entry for
65      // the context, otherwise, it returns a weight of 0.
66      weight = function(context) {
67        if (weights[context] != null) {
68          return weights[context];
69        } else {
70          return 0;
71        }
72      };
73
74      // Base case: if the current context has no children, return
75      // the node
76      if ((children == null) || !(children[node] instanceof Array)) {
77        return node;
78      }
79
80      // Construct a list of {node, weight} pairs for each of the
81      // children of the current node.
82      childWeights = (function() {
83        var _l, _len3, _ref3, _results;
84        _ref3 = children[node];
85        _results = [];
86        for (_l = 0, _len3 = _ref3.length; _l < _len3; _l++) {
87          c = _ref3[_l];
88          _results.push({
89            'node': c,
90            'weight': weight(c)
91          });
92        }
93        return _results;
```

```
94     })();
95
96     // Find the child weight the highest weight
97     maxWeight = _.max(childWeights, function(w) {
98       return w.weight;
99     });
100
101    // If that weight is sufficiently high, continue searching
102    // in that node.
103    // Else, return the current node as overarching.
104    if (maxWeight.weight > minimum) {
105      return findContext(maxWeight.node, weights, minimum);
106    } else {
107      return node;
108    }
109  };
110
111  // ...
112
113});
```

In the piece of code that follows, we listen to changes in the search form. If the form is submitted, we query the ElasticSearch index with the given query. The given query is matched against a number of field in the index:

- skos:prefLabel,
- skos:definition,
- title,
- content,
- concerns_readable,
- context_readable

Some of these fields (such as the title of a document) are given a higher 'importance' than others. If the search terms appear in the title, this will result in a high search score for the document.

```
1 // Attach a callback to the form/submit event
2 $form.submit(function(e) {
3   // Prevent the forms default behavior: loading another page
4   e.preventDefault();
5
6   // Enable the progress indicator again
```

```
 7    $results.addClass('csspinner');
 8
 9    // Search ElasticSearch for the query.
10    // again, this query returns a promise the will be
11    // resolved asynchronously
12    // Note that the '^3' signs in the field-list indicate
13    // a weight of 3 times the normal weight.
14    resultsPromise = client.search({
15      index: 'hzbwnature',
16      type: 'intentional_element',
17      size: 100,
18      body: {
19        query: {
20          multi_match: {
21            query: $field.val(),
22            fields: [
23              "skos:prefLabel^3",
24              "skos:definition",
25              "title^3",
26              "content",
27              "concerns_readable^2",
28              "context_readable^2"
29            ]
30          }
31        }
32      }
33    });
```

And lastly, there is the block of code that is called when the search results are available.

```
 1 resultsPromise.then(function(resp) {
 2
 3   // Declare a bunch of variables
 4   var c, context, hit, hits, minimum, storage, _l,
 5          _len3, _len4, _len5, _m, _n, _ref3;
 6
 7   // Log the response to the browser console
 8   console.log(resp);
 9
10   // Get the actual hits from the response object
11   hits = resp.hits.hits;
12
13   // Empty the results area of the page and remove the spinner
14   $results.html("");
15   $results.removeClass('csspinner');
16
```

```
17   // For each hit, add it to the list of search results.
18   for (_l = 0, _len3 = hits.length; _l < _len3; _l++) {
19     hit = hits[_l];
20     $results.append($('<div>').html(result_template(hit)));
21   }
22
23   // Now add the scores of the hits to their contexts (recursively)
24   // using the above described functions
25   storage = {};
26   for (_m = 0, _len4 = hits.length; _m < _len4; _m++) {
27     currentHit = hits[_m];
28
29     // Loop through the contexts of the current hit and attach weight
30     _ref3 = currentHit._source.context;
31     for (_n = 0, _len5 = _ref3.length; _n < _len5; _n++) {
32       c = _ref3[_n];
33       storage = addWeight(c, currentHit._score, storage);
34     }
35   }
36
37   // Define the minimum weight (an arbitrary choice now)
38   // and find the overarching context
39   minimum = storage.ROOT * 0.8;
40   context = findContext('ROOT', storage, minimum);
41
42   // Show the overarching context on the page.
43   $('.context').text(context);
44
45 });
```

This section contained quite some code. Hopefully, this was helpful to attain a deep understanding of how searching with ElasticSearch and finding the overarching context—extracting a bit of meaning—works.

Unfortunately, the spread activation technique that is to be the keystone in our search method could not be implemented yet. We plan to do this in the break, when there is enough training data available.

5 | Conclusion

The aim of this project was always to "become better than Google". Although this is of course an optimistic goal, it is definitely not impossible. By a hybrid combination of semantic methods that have proven to perform well, and a thesaurus that is curated by professionals instead of automatically generated, we hope to perform better than Google on very specialized fields. We expect that spread activation will give our method a high recall, while restricting the content to a semantic context assures high precision of the results.

We can not yet assess the performance of our search method, since it is not fully implemented yet, and not enough training data is available at the moment. The searching as available now is, however, already useful and an improvement to MediaWiki's built in search.

The next steps would be to (1) implement spread activation and (2) thoroughly test the performance of the method with real life data. At the moment, some parameters of the method, such as the minimum percentage of weight that the 'search context' should contain, are chosen fairly arbitrarily. These values have to tuned continuously in the future.

Another area in which this work can grow is in the field of machine learning. It is a key feature of our product that relations are curated by professionals, but we should aid these professionals by at least suggesting tags and relations for and between entities. A learning algorithm could, for instance, be used to provide automatic SKOS tags for pages based on statistical models. Also, we should learn from the search queries of future users to continuously improve our service.

Furthermore, the users should be empowered to control their own search results. Although we identify a search context, we should allow a future user to change this default if it is incorrect. The context could also depend on the area of the website the user is searching from.

All in all, this has been a fruitful project, and I look forward wrapping up this project and transferring obtained knowledge this summer.

References

[1] Allemang, D., and J.A. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Waltham, MA: Morgan Kaufmann/Elsevier, 2011. Print.

[2] Koren, Y. *Working with MediaWiki.* WikiWorks Press, 2012. Print.

[3] Akrivas, Giorgos, Manolis Wallace, Giorgos Andreou, Giorgos Stamou, and Stefanos Kollias. "Context-sensitive semantic query expansion." In *Artificial Intelligence Systems*, 2002.

[4] Knappe, Rasmus, Henrik Bulskov, and Troels Andreasen. "Perspectives on ontology-based querying." *International Journal of Intelligent Systems* 22, no. 7 (2007): 739–761.

[5] Lee, Ming-Che, Kun Hua Tsai, and Tzone I. Wang. "A practical ontology query expansion algorithm for semantic-aware learning objects retrieval." *Computers & Education* 50, no. 4 (2008): 1240–1257.

[6] Qiu, Yonggang, and Hans-Peter Frei. "Concept based query expansion." In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 160–169. ACM, 1993.

[7] Rocha, Cristiano, Daniel Schwabe, and Marcus Poggi Aragao. "A hybrid approach for searching in the semantic web." In *Proceedings of the 13th international conference on World Wide Web*, pp. 374–383. ACM, 2004.

[8] Checkland, Peter B. "Soft systems methodology." *Human systems management* 8, no. 4 (1989): 273–289.

[9] Voorhees, Ellen M. "Query expansion using lexical-semantic relations." In *SIGIR'94*, pp. 61–69. Springer London, 1994.

[10] Hoang, Hanh Huu, and A. Min Tjoa. "The state of the art of ontology-based query systems: A comparison of existing approaches." In *Proceedings of ICOCI06* (2006).

[11] Srinivasan, Naveen, Massimo Paolucci, and Katia Sycara. "An efficient algorithm for OWL-S based semantic search in UDDI." In *Semantic Web Services and Web Process Composition*, pp. 96–110. Springer Berlin Heidelberg, 2005.

Acknowledgements

Thanks go to my supervisors, Hans de Bruin and Henk Meijer for initiating a connection between the Hogeschool Zeeland and University College Roosevelt that made this project possible. I believe the opportunity to do actual applied research is a valuable addition to the research track at UCR. Through this project, I learned that the problems faced in this type of work are very different from academic issues and that they are definitely something worth experiencing at university. Another round of thanks go to Hans de Bruin, Wouter Everse and Anton Bil (HZ). You have been very pleasant colleagues and I have been looking forward to our weekly meetings in Vlissingen throughout the project.