Master Thesis

A User Interface for Semantic Full Text Search

Florian Bäurle

July 2011

University of Freiburg

Faculty of Engineering

First Reviewer:	Prof. Dr. Hannah Bast
Second Reviewer:	Prof. Dr. Andreas Podelski
Supervisors:	Prof. Dr. Hannah Bast, Björn Buchhold

Declaration

I hereby declare that I am the sole author and composer of this thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that this thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Florian Bäurle

Abstract

We present a powerful and comfortable interactive web user interface for semantic full-text search. It combines and extends established components of other search user interfaces - namely keyword search, facets, proposals and breadcrumbs - in a new way. The result is a reasonably easy to use semantic search application that assists its users in an incremental creation of semantic queries, without requiring them to have knowledge about any special query language or the underlying ontology.

The user interface provides only a single input field as it is also common for current keyword search applications like GOOGLE. When something is typed, it automatically shows different proposals for words, relations, entities, or semantic classes that are used to build the search query. When any of the proposals is selected, it is added to the query which is displayed as a tree in an advanced breadcrumbs panel. With the help of the breadcrumbs panel, the query can easily be extended or changed by adding new proposals or replacing or deleting already existing parts. The provided proposals are context sensitive with respect to the current query to support users in the incremental creation of meaningful queries. The hits for the current query are always automatically displayed together with evidence why they suit to the query.

In this work we address the design decisions that were made during the implementation of the interface application and explain its basic architecture. We also show what queries are supported by the user interface and how it is used to create them.

Zusammenfassung

Wir präsentieren eine leistungsfähige und komfortable interaktive Web-Benutzeroberfläche für semantische Volltextsuche. Sie kombiniert und erweitert etablierte Bestandteile von anderen Such-Benutzeroberflächen - und zwar Schlüsselwort-Suche, Facetten, Vorschläge und Brotkrumen - auf eine neue Art und Weise. Das Ergebnis ist eine relativ einfach zu verwendende semantische Such-Applikation, die ihre Benutzer bei der schrittweisen Erstellung von semantischen Suchanfragen unterstützt, ohne dass eine spezielle Abfragesprache oder die zugrundeliegende Ontologie bekannt sein muss.

Die Benutzeroberfläche verfügt nur über ein einziges Eingabefeld, wie es für gewöhnliche aktuelle Suchanwendungen wie Google ebenfalls üblich ist. Wenn etwas eingetippt wird zeigt sie automatisch verschiedene Vorschläge für Wörter, Relationen, Entitäten oder semantische Klassen an, welche verwendet werden, um die Suchanfrage zu erstellen. Wenn einer der Vorschläge ausgewählt wird, dann wird er zur Suchanfrage, die als Baum in einem erweiterten Brotkrumen-Panel dargestellt wird, hinzugefügt. Anhand des Brotkrumen-Panels kann die Suchanfrage leicht ergänzt oder abgeändert werden, indem neue Vorschläge hinzugefügt oder bereits vorhandene Teile ersetzt oder entfernt werden. Die angebotenen Vorschläge sind kontextabhängig von der aktuellen Suchanfrage, um Benutzer bei der schrittweisen Erstellung von sinnvollen Suchanfragen zu unterstützen. Die Treffer für die aktuelle Suchanfrage werden immer automatisch zusammen mit einem Nachweis, weshalb sie zur Suchanfrage passen, angezeigt.

In dieser Arbeit gehen wir auf die Designentscheidungen ein, die im Laufe der Implementierung der Anwendung getroffen wurden, und erklären ihre grundlegende Architektur. Wir zeigen auch welche Suchanfragen von der Benutzeroberfläche unterstützt werden und wie sie benutzt wird, um diese zu erstellen.

Acknowledgments

First of all I would like to thank my supervisor, Hannah Bast, who made this thesis possible by providing the topic. Furthermore I want to thank her for the guidance and support without which I would not have been able to complete this thesis. This also includes the many hours of meetings we had together with Björn Buchhold to constantly evaluate and improve the user interface application from the initial concept to its current state. I also want to thank Björn for his constant support with the search back end and the valuable tips he gave me. In addition, my thanks go to Andreas Podelski for taking the role of the second reviewer.

I also want to thank Benjamin Bäurle, Tim Kraski and Gönke-Britt Hansen for investing their time to help me a lot by reviewing this document. Another special thanks goes to Daniel Dietsch for his advice on scientific writing and literature management.

At a personal level, I would like to thank my parents. Their steady support made everything possible in the first place.

Contents

1. Introduction

1.1. Motivation

Search engines like GOOGLE, YAHOO or BING are very popular because they are easy to use and achieve good results for everyday common search tasks. Basically they take queries with a number of keywords and then return results that contain all or some of these keywords. But this is also a limitation of these search engines because they can only find documents that contain the keywords literally, or maybe also synonyms of these keywords. They do not have the ability to understand the semantics of a query. When, for example, searching for the keyword "*scientist*", they can only find documents that contain this word. They cannot find documents that only contain the name of a scientist, like, for example, Albert Einstein, even if this was intended.

Let us consider a more intricate query like "Movies directed by Steven Spielberg that are about one of the world wars". The intention of the query could be to just find the names of such movies or to find documents about such movies, like, for example, reviews. Therefore a search engine must be able to understand the semantics of the query. For the example query this means that we want to find instances of the class of movies instead of the literal word *movie*. Furthermore the words *"directed by"* mean a relation that narrows down the movies to those that were directed by the entity *Steven Spielberg*, which is an instance of the class of directors. And at last, the movies have to be somehow related to one of the world wars.

Apart from the previously described search engines, there also exist search engines that can process semantic information in queries. One kind of semantic search engines is solely focused on querying ontologies [Corby 06, Neumann 08, Broekstra 03]. Such search engines have been tested to be quite powerful and efficient, even for large data sets, but they are limited to the retrieval of facts from their underlying ontologies. This implies for our example query that these search engines would only be able to provide facts like the name, the release date, or the running time of such movies, depending on the content of their ontologies. There are also other kinds of semantic search engines that bypass this limitation by combining ontology and full text search in different ways [Bast 07, Buchhold 10, Hahn 10, Waitelonis 11]. Thus they make it possible to search in document collections using semantic queries. This allows them to provide much more types of results than only facts, even for semantic queries. For instance reviews or trailers of movies matching the sample query, depending on the search content of the search engines.

One challenge in the development of semantic search engines is the query creation. The queries for these engines must be expressed in a format that enables them to understand their semantic meaning, so that they are able to answer them. The probably most intuitive and convenient way for humans to formulate queries is to express them in natural language sentences. Unfortunately, due to the complexity and ambiguities of natural languages, natural language processing is a very complicated and difficult task [Di Martino 10, Gao 11]. For this reason, special query languages like SPARQL[1] were developed to define semantic queries for the search in ontologies. These query languages

[1] SPARQL is a query language for the Resource Description Framework (RDF) by the W3C. [W3C 11c]

perform great in describing complex queries. However, you do not only need to know the syntax of the language, but you also need to be familiar with the underlying ontologies to be able to formulate correct queries. This is a barrier for end users which is not acceptable if a semantic search engine is intended to be available to the masses. Hence there is a need for comfortable user interfaces that hide these details without limiting the capabilities of the search engines by only providing a limited set of features. In this thesis, we present a new semantic search user interface that we developed to meet these requirements.

1.2. Contributions

This thesis only covers our new user interface for semantic search. In order to implement and test it, we use a prototype of the semantic search engine that we are currently developing under the working title Broccoli as the back end. The prototype is basically an extended version of Susi (see section 2.1) that has some new functionality. The back end is necessary to get the proposals, which are used to create the queries with the user interface, as well as to get the displayed results. In the future, the back end will be replaced by a newly implemented search engine to improve the results and the query times.

1.3. Structure of the Thesis

The rest of the document is organized as follows:

- Chapter 2 shows related work and briefly points out some differences to our approach.
- In chapter 3 we explain the content in which we search and what queries are supported by the user interface.
- Chapter 4 presents the user interface. It describes its layout, what features it offers and explains how it is used.
- Chapter 5 deals with the realization of the user interface. This includes the reasons for choosing Google Web Toolkit as a framework, the overall architecture, and some further implementation details.
- In chapter 6 we discuss our conclusions and possible future work.

Rdf is a standard model for data interchange on the Web by the W3C. It extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”). [W3C 11b]

2. Related Work

There already exist some other approaches for semantic search user interfaces that do not rely on natural language processing. An add-on for DSpace[2] [Koutsomitropoulos 11] offers, for example, a user interface that assists users in the construction and submission of queries to the DSpace ontology. The interface automatically disallows combinations that make no sense and offers auto-completion choices for supplied input to build up query expressions in Manchester Syntax[3]. The interface exposes information about the ontology through the auto completion feature and checks for a valid syntax, but since it just assists in building up a formal query string, at least a basic understanding of its syntax is still required to use the application.

Other applications like SemSearch [Lei 06] or Spark [Zhou 07] take another approach to provide semantic search user interfaces that hide the complexity of semantic search, such as the structure of the ontology or special formal query languages, to be suitable for ordinary end users. Their interfaces only have a single input field which they use to automatically translate keyword queries into formal semantic query languages. This makes them similarly easy to use as search interfaces like Google. The drawback of this approach is that the quality of the results depends more on the query interpretation efficiency than the ability of the users to formulate exact queries. To improve this issue, Spark presents the users a list of possible query interpretations from which they must choose what they meant, but the quality of this list and its ranking still remain out of the control of the users.

Then there exist facet browsing approaches like /facet [Hildebrand 06] or Faceted Wikipedia Search [Hahn 10]. They provide an intelligent facet navigation that adapts to the previously selected facets, so that only relevant ones are displayed. Selecting a facet from the navigation adds it to the list of active ones that represent the query and reduces the result set to all items that fulfill it. Therewith users can conveniently navigate through data collections. However this approach does not allow for complex queries because it is not possible to model dependencies between selected facets. Contentus [Waitelonis 11] is a search user interface that combines keyword search and semantic facet browsing. It starts with the search for a keyword and then provides semantic facets to further restrain the found results. The main difference to the other mentioned facet based user interfaces is that Contentus does not only use facts of its underlying ontology, but also performs full-text search on articles and transcripts of audio and video files. Additionally they have a user management for their interface that makes it possible to provide personalization features. Currently this is limited to an optional collection of personal queries, but they plan to expand this in the future. For instance, they want to add the possibility to connect with friends to be able to share results with them.

[2] DSpace is a digital repository system that captures, stores, indexes, preserves and redistributes research material in digital formats. It can be used for a variety of digital archiving needs ranging from institutional repositories to learning object repositories or electronic records management, and more. DSpace is freely available as open source software. [DSpace 11]

[3] The Manchester Syntax is a syntax which is designed for writing Owl class expressions [CO-ODE 11]. The Owl Web Ontology Language is an ontology language for the semantic web by the W3C [W3C 11a].

A further different approach is pursued by Ester [Bast 07]. Ester also uses an ontology for the semantic information but utilizes it to provide semantic full-text search on the English Wikipedia. The user interface for the search has a single keyword input field. Semantic information is handled by automatically detecting semantic classes for entered keywords and providing different proposal boxes that help to disambiguate classes and offer specific entities of these classes to refine the query.

The following sections of this chapter describe those user interfaces that we found to be most relevant with respect to our new implementation in more detail. We also address differences to the approach of this thesis.

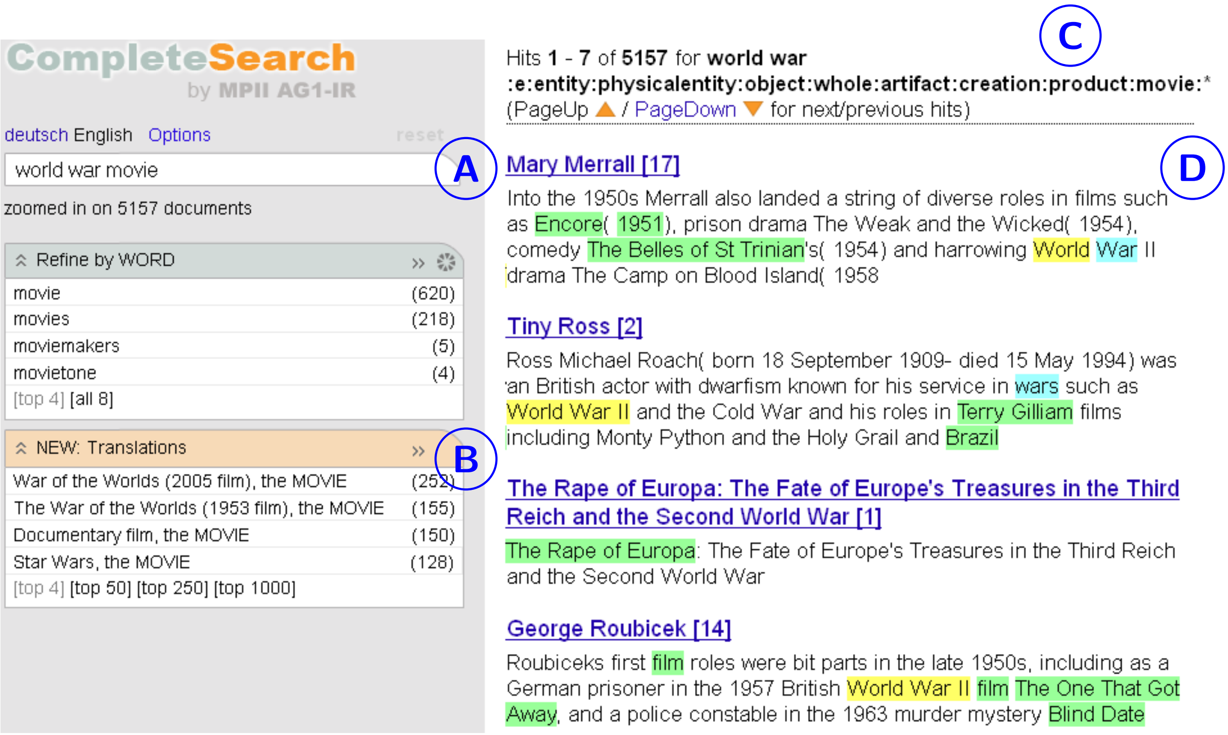
2.1. ESTER & SUSI

The user interfaces of Ester [Bast 07] and Susi [Buchhold 10] are probably the most similar related work to this thesis, as they were designed for the same type of semantic full-text search. Strictly speaking, the new user interface that we present was even designed for the further development of their search engines.

Ester is a search engine that combines full-text and ontology search using only two basic operations: prefix search and joins. This allows for very fast query processing and a fully interactive and proactive user interface, which is very similar to the interface of Susi that is shown in figure 1 and explained in more detail at the end of this section. As a proof of concept, Ester was applied to the English Wikipedia combined with the Yago[4] ontology. Its user interface is proactive in the way that it automatically reacts on user input using JavaScript, without needing the user to press a search button. Ester uses this to automatically detect semantic classes when they are typed into its input field. Therefore it checks after every keystroke if the last prefix of the query string matches a class name, and if more than one class matches, it displays a small box on the left-hand side with proposals for all matching classes. These proposals can then be clicked to replace the prefix in the query string with the class name. Below this box there is a second proposal box that displays instances of the last class in the query string which can be clicked to restrain the query to a single entity instead of the complete class. On the right-hand side Ester displays the results to the current query. Those are also automatically updated when the query string is changed to always show the results to the current query. Ester has no special breadcrumbs panel or anything similar that provides the user with a visual aid how the query string is actually exactly handled. Entered keywords can be interpreted as semantic classes or simple word occurrences without a visible difference in the user interface. Furthermore, the mentioned proposal boxes for available classes or entities only display results for the last keyword that was appended to the query string. This makes query refinements quite cumbersome and unintuitive because the order of the keywords in the query string has to be changed if users want proposals for other parts of the query. These are usability issues that are addressed by the new user interface which is presented in this thesis.

Susi [Buchhold 10] can basically be considered as a successor of Ester. It aims for the same results, combining full-text and ontology search, and using the same combination

[4] Yago is a huge semantic knowledge base, derived from Wikipedia and WordNet. [MPII 11]



CompleteSearch by MPII AG1-IR

deutsch English Options reset

world war movie

zoomed in on 5157 documents

Refine by WORD

movie	(620)
movies	(218)
moviemakers	(5)
movietone	(4)

[top 4] [all 8]

NEW: Translations

War of the Worlds (2005 film), the MOVIE	(252)
The War of the Worlds (1953 film), the MOVIE	(155)
Documentary film, the MOVIE	(150)
Star Wars, the MOVIE	(128)

[top 4] [top 50] [top 250] [top 1000]

Hits **1 - 7** of **5157** for **world war :e:entity:physicalentity:object:whole:artifact:creation:product:movie:***
(PageUp ▲ / PageDown ▼ for next/previous hits)

Mary Merrall [17]

Into the 1950s Merrall also landed a string of diverse roles in films such as Encore(1951), prison drama The Weak and the Wicked(1954), comedy The Belles of St Trinian's(1954) and harrowing World War II drama The Camp on Blood Island(1958

Tiny Ross [2]

Ross Michael Roach(born 18 September 1909- died 15 May 1994) was an British actor with dwarfism known for his service in wars such as World War II and the Cold War and his roles in Terry Gilliam films including Monty Python and the Holy Grail and Brazil

The Rape of Europa: The Fate of Europe's Treasures in the Third Reich and the Second World War [1]

The Rape of Europa: The Fate of Europe's Treasures in the Third Reich and the Second World War

George Roubicek [14]

Roubiceks first film roles were bit parts in the late 1950s, including as a German prisoner in the 1957 British World War II film The One That Got Away, and a police constable in the 1963 murder mystery Blind Date

Figure 1: Example screenshot for the user interface of SUSI. The **A** marks the input field. Classes are automatically detected for the last input word. Since the proposal boxes below the input field always display proposals for the last prefix in the query string, the entity proposal box, which is marked by **B**, shows instances for the automatically detected movie class. The **C** marks the query string that the user interface generates to get the results. As can be seen, the movie class is represented by a very long obscure word. Classes and Entities can also be defined manually in the query string by using such special words. The hits are displayed together with evidence on the right-hand side as marked by **D**.

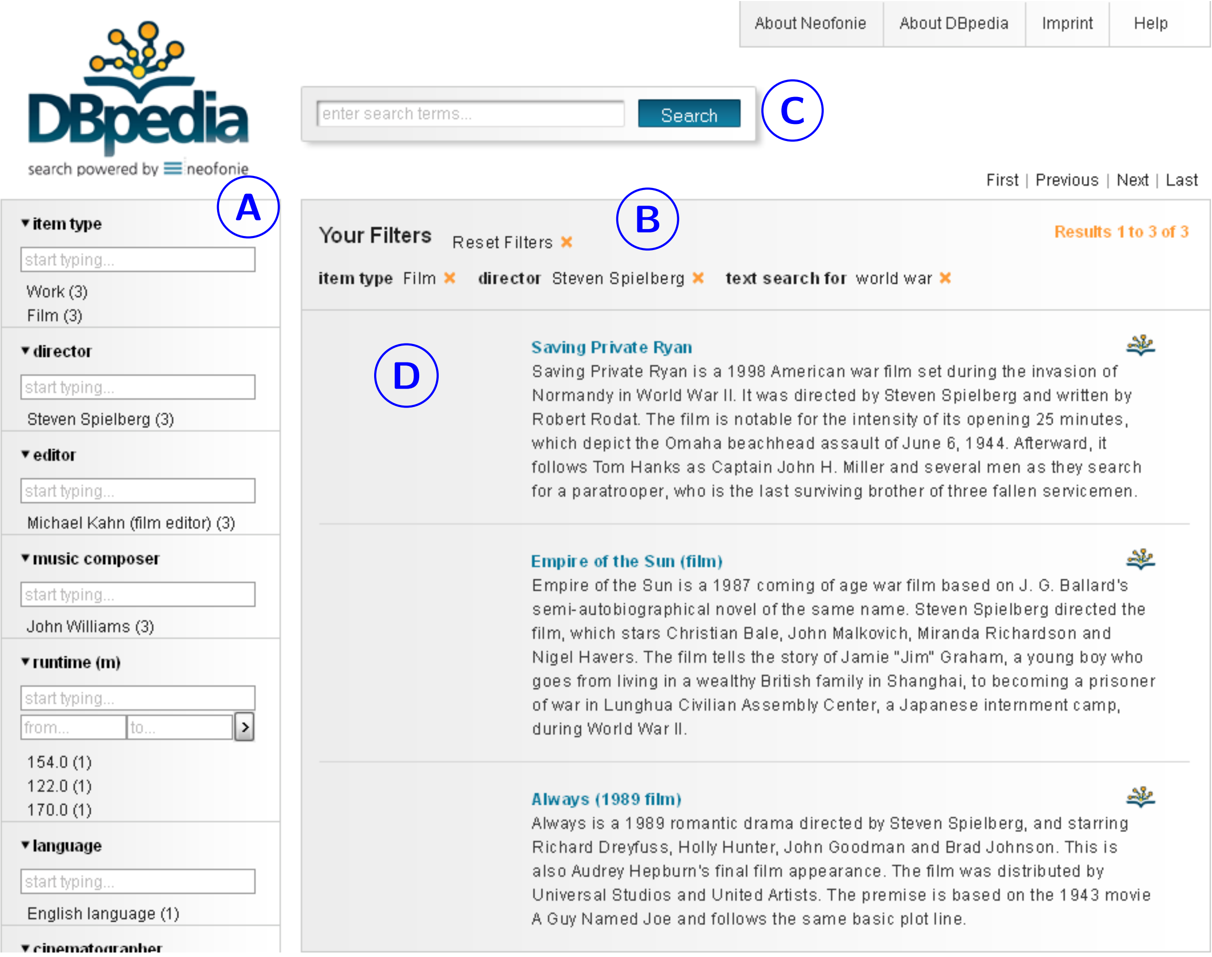
of the English Wikipedia and YAGO. We do not want to go into detail about all the differences between SUSI and ESTER as these are not relevant here. The main difference is that SUSI further reduced the needed operations to prefix search only, to be even faster than ESTER. This is achieved by limiting the relations to the taxonomy of the classes and entities of the ontology. Another relevant difference is that SUSI treats each sentence of a Wikipedia article as a separate document instead of treating the whole article as a single document. This is done to improve the precision. As already stated in section 1.2, a modified version of SUSI is currently used as a prototype for the back end of the new user interface, so this document handling reflects in the results of later examples. For testing, SUSI provides a user interface that is very similar to that of ESTER and has the same drawbacks. Figure 1 shows a screenshot of the interface. Since it is not possible to formulate the sample query from the introduction with it, the query in the screenshot only tries to find movies about one of the world wars. Neither Ester nor Susi support the precise creation of more complex queries that would require a combination of relations and full-text. Even if they would support more complex queries, it would require a quite obscure syntax to express them in the single input field. Building slightly more complex queries than the one in the screenshot is already very user unfriendly because of the needed obscure words (e.g. the string for the word class marked by the C in figure 1) to describe further classes or entities.

2.2. Faceted Wikipedia Search

Faceted Wikipedia Search [Hahn 10] is a sophisticated search user interface to query the DBpedia[5] ontology. It is actually one of the few state of the art user interfaces we found. It has an appealing professional design and does not appear overloaded. For this reason, the layout of our new user interface is loosely oriented on that design. But even though their layout is basically similar, they fairly differ in their functionality. While our user interface is designed for semantic full-text search, Faceted Wikipedia Search is designed as a facet browser. Figure 2 shows a screenshot of the interface trying to find results for the example query of the introduction, what is only conditionally working. With the help of the available facets one can easily query movies that were directed by Steven Spielberg, but there is no facet type that can be used to limit these movies to those that are about one of the world wars. Using the keyword search function to only get world war movies only yields very few results because the words “world war” do not appear in many article abstracts.

Using the Faceted Wikipedia Search user interface, queries are constructed by selecting facets from the boxes on the left-hand side of the interface. The types of available facets intelligently adapt to the previously selected ones. Nevertheless, it can be cumbersome to find the desired type because not all of the usually many available facet types are displayed by default. To add more, a user has to click a separate button. To avoid this we limit the number of proposal boxes in our application to just four, which are always displayed. Additionally, each facet type has its own input field to search for a certain facet of that type. Our approach gets along with only a single input field that is used to filter all of our proposal boxes. Their application also provides an input field for text search, but it is only used for a simple keyword search in the abstracts of the Wikipedia articles that are selected through the facets and not even in their full text. The Faceted Wikipedia Search application does not show evidence for the returned results. It just presents a list of Wikipedia articles that are directly chosen through the selected facets. Our full-text search user interface shows evidence so that the users are able to comprehend why the results were returned. Another difference is that our interface is more interactive and gives more feedback to the user. When the query is changed, for example by adding a new facet, their user interface does not respond until it is reloaded with the new results. This can be confusing because query changes for their search currently take several seconds in which it is not signaled to the user that the application is working. Our user interface displays loading animations to signal its state to the user whenever it is loading new data asynchronously to not block the application. Another similarity is their breadcrumbs display. The Faceted Wikipedia Search user interface has a panel that shows

[5] DBpedia is a community effort to extract structured information from Wikipedia and to make this information available on the Web. It allows you to ask sophisticated queries against Wikipedia, and to link other data sets on the Web to Wikipedia data. [DBpedia 11]



▾ item type
start typing...
Work (3)
Film (3)

▾ director
start typing...
Steven Spielberg (3)

▾ editor
start typing...
Michael Kahn (film editor) (3)

▾ music composer
start typing...
John Williams (3)

▾ runtime (m)
start typing...
from... to...
154.0 (1)
122.0 (1)
170.0 (1)

▾ language
start typing...
English language (1)

▾ cinematographer

Your Filters Reset Filters ✕ Results 1 to 3 of 3

item type Film ✕ director Steven Spielberg ✕ text search for world war ✕

Saving Private Ryan
Saving Private Ryan is a 1998 American war film set during the invasion of Normandy in World War II. It was directed by Steven Spielberg and written by Robert Rodat. The film is notable for the intensity of its opening 25 minutes, which depict the Omaha beachhead assault of June 6, 1944. Afterward, it follows Tom Hanks as Captain John H. Miller and several men as they search for a paratrooper, who is the last surviving brother of three fallen servicemen.

Empire of the Sun (film)
Empire of the Sun is a 1987 coming of age war film based on J. G. Ballard's semi-autobiographical novel of the same name. Steven Spielberg directed the film, which stars Christian Bale, John Malkovich, Miranda Richardson and Nigel Havers. The film tells the story of Jamie "Jim" Graham, a young boy who goes from living in a wealthy British family in Shanghai, to becoming a prisoner of war in Lunghua Civilian Assembly Center, a Japanese internment camp, during World War II.

Always (1989 film)
Always is a 1989 romantic drama directed by Steven Spielberg, and starring Richard Dreyfuss, Holly Hunter, John Goodman and Brad Johnson. This is also Audrey Hepburn's final film appearance. The film was distributed by Universal Studios and United Artists. The premise is based on the 1943 movie A Guy Named Joe and follows the same basic plot line.

Figure 2: Example screenshot for the Faceted Wikipedia Search user interface. The **A** marks the boxes with the available facet types. From these, concrete facets can either be directly selected or manually entered through the input fields. Selecting a facet adds it to the breadcrumbs panel which is marked by **B**. It displays the selected filters that form the query. The input field that is marked by **C** can be used to add an additional keyword search filter for the article abstracts. The **D** marks the displayed resulting Wikipedia articles.

all selected facets and allows to delete them independently from each other. We also have a breadcrumbs panel in our interface. The main difference is that our breadcrumbs are not independent from each other, but they form a tree that depicts the meaning of the created query. Accordingly, our breadcrumbs panel offers more sophisticated refinement possibilities for the query than just removing single parts. The last thing to mention is that their application displays an image of the Wikipedia articles in the results, if possible. Since our application currently also searches in Wikipedia articles, we also added such image thumbnails to the results in our user interface, because they are a great help in the evaluation of the results.

3. Search Content and Queries

In the previous chapters we presented a motivation why further research in the area of semantic search user interfaces is needed and we presented some related work that was already done. Before we go into details about the functionality and implementation of our new user interface, this chapter explains the search content and the query language for which our application was designed. Hildebrand et al. [Hildebrand 07] have already analyzed that these have a substantial impact on the design of the user interface for a search engine.

3.1. Search Content

Every search engine is designed for searching in a special type of data. We call this type of data the search content. Some engines are designed for searching in websites, others search for facts in ontologies, find information about scientific papers, or are built to search the contents of a library, just to name a few examples. The search content has a crucial impact on the user interfaces for search engines because it dictates the kind of information that needs to be displayed to the user. Furthermore, the possibilities to navigate through the results or for query refinements also depend on it.

For a concrete example of the impact of the search content one can compare the user interfaces of the GOOGLE web search[6] and the earlier mentioned CONTENTUS. The structure of the GOOGLE user interface is quite simple. It only has one keyword input field and a list of results that contains links to the found websites together with a small text excerpt as evidence. In contrast to that, the CONTENTUS user interface is far more complex. It was designed for the search in archives of museums and libraries that contain books, images, tapes and films. Accordingly, its user interface does not only have a single input field, but also a multitude of selectable facets that are displayed in an additional breadcrumbs panel when they are activated. The results do not only show a small text excerpt as evidence, but offer special views that allow to navigate through found videos or audio files. Additionally, the results can be further explored, for instance, via the use of displayed related entities that are obtained through an underlying ontology.

Our new user interface is designed for a semantic full-text search engine which searches in a text collection that is linked with an ontology. The back end prototype that is currently used for the user interface uses the English Wikipedia as text collection and YAGO as linked ontology, like already pointed out in section 2.1. Consequently, our new user interface needs to be able to display facts from an ontology as well as text documents for the results. How this is done is shown in chapter 4.

3.2. Query Language

The query language describes the type of information that is needed to define a query for the search engine and how it is structured. The queries that can be created with our new

[6] `http://www.google.com/`

user interface can be described as trees. Hence we define the set of possible query trees with a regular tree grammar, based on [Mani 03].

A regular tree grammar is a 4-tuple $G = (N, T, S, P)$, where:

- N is a finite set of non-terminals,
- T is a finite set of terminals,
- S is a set of start symbols, where $S \subset N$,
- P is a finite set of production rules of the form $X \rightarrow a\,[RE]$, where $X \in N$, $a \in T$, and RE is a regular expression over N.

The nodes of our query trees can have 6 different types: class, entity, cooccurrence (abbr.: coocc), relation, value and word. The different types of nodes denote the following values:

Class	A semantic class from the ontology.
Entity	An instance of a class from the ontology.
Coocc	The cooccurrence of a class or entity with other classes, entities or words.
Relation	A relation between classes or entities that is defined in the ontology.
Value	A numerical value, e.g. a date, from the ontology.
Word	A combination of letters that form a single or multiple words.

So we define the tuples of our regular tree grammar as:

$N = \{CLASS, ENTITY, COOCC, RELATION, VALUE, WORD\}$

$T = Classes \cup Entities \cup \{\text{occurs-with}\} \cup Relations \cup Values \cup Words$, where $Classes$, $Entities$, $Relations$, $Values$ and $Words$ respectively denote the sets of possible class, entity, relation, value and word nodes, and the element occurs-with denotes a cooccurrence node.

$S = \{CLASS, ENTITY, WORD\}$

$$
\begin{aligned}
P: \quad CLASS &\rightarrow class\,[(COOCC|RELATION)^*] \\
ENTITY &\rightarrow entity\,[(COOCC|RELATION)^*] \\
COOCC &\rightarrow \text{occurs-with}\,[WORD\,(CLASS|ENTITY|WORD)^*] \\
RELATION &\rightarrow relation\,[CLASS|ENTITY|VALUE] \\
VALUE &\rightarrow value\,[\epsilon] \\
WORD &\rightarrow word\,[\epsilon] \qquad ,
\end{aligned}
$$

where $class \in Classes$, $entity \in Entities$, $relation \in Relations$, $value \in Values$ and $word \in Words$.

A production rule $X \rightarrow a\,[RE]$ denotes that the non-terminal X can generate a subtree with a as the root and with children that match RE. This means that class and entity nodes can either be leaves or that they can form a subtree with an arbitrary number of cooccurrence and relation children. A cooccurrence node always creates a subtree, since it must have at least one word child[7] and can have arbitrary further class, entity, or word child nodes. Note that all child nodes of a cooccurrence node must occur together in the same document with the parent node. If this is not desired, one can add multiple different cooccurrence nodes to the parent node, these are then independent of each other. Relation nodes also always create a subtree since they must have exactly one class, entity, or value node as their target child. Value and word nodes cannot have any children and thus may only occur as leaves.

The set of trees that can be generated from any start symbol forms the language generated by a regular tree grammar. So we can write our query language as:

$$L = \{t \,|\, t \text{ is a tree that was generated with } G\}$$

Let us have a look at an example. In the introduction we mentioned the following sample query:

Example Query 1. *"Movies directed by Steven Spielberg that are about one of the world wars"*

We want to search for the class of *movies*, so we take it as the root. Then we narrow it down by adding the *directed-by* relation and set the entity *Steven_Spielberg* as its target. Furthermore we only want to get movies about one of the world wars, so we further narrow down the *movie* class by adding that it must co-occur with the words *world war*. This is possible because we linked the ontology with a text collection for our search content. When we supply our tree grammar only with the identified concrete terminal values we get the grammar $G_1 = \{N, T_1, S, P_1\}$, with:

$$\begin{aligned} T_1 = \;& \{\text{CLASS:movie}, \text{ENTITY:Steven\_Spielberg}, \text{occurs-with}, \text{RELATION:directed-by}, \\ & \text{WORD:world war}\} \end{aligned}$$

$$\begin{aligned} P_1: \quad \mathit{CLASS} &\rightarrow \text{CLASS:movie}\,[(\mathit{COOCC}|\mathit{RELATION})^*] \\ \mathit{ENTITY} &\rightarrow \text{ENTITY:Steven\_Spielberg}\,[(\mathit{COOCC}|\mathit{RELATION})^*] \\ \mathit{COOCC} &\rightarrow \text{occurs-with}\,[\mathit{WORD}\,(\mathit{CLASS}|\mathit{ENTITY}|\,\mathit{WORD})^*] \\ \mathit{RELATION} &\rightarrow \text{RELATION:directed-by}\,[\mathit{CLASS}|\mathit{ENTITY}|\,\mathit{VALUE}] \\ \mathit{WORD} &\rightarrow \text{WORD:world war}\,[\epsilon] \end{aligned}$$

Using grammar G_1 we can generate the tree in figure 3, which expresses the example query number 1.

Our query tree grammar allows a word node as the root, though a word node can only be a leaf and must not have any children. This is allowed to also enable normal keyword search over the text collection of the search content. Thus, queries with only a word

[7] The reason that a cooccurrence node must have at least one word child node is a restriction of the current back end prototype. Due to its implementation, it needs at least one word to be able to handle a cooccurrence efficiently. This restriction is currently acceptable because the cooccurrence of two things mostly has a certain connection. This connection can be expressed by the needed word. This restriction might be lifted in the future when the back end gets replaced by a new implementation.

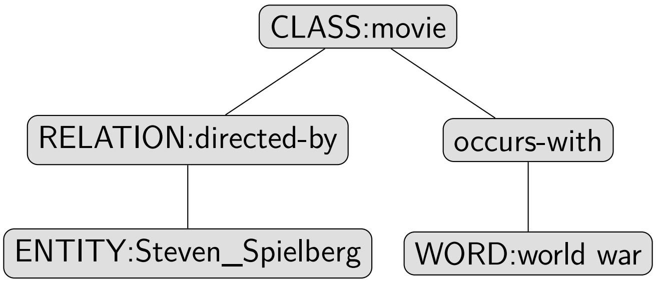


Figure 3: Sample query tree for the query "Movies directed by Steven Spielberg that are about one of the world wars".

node as the root express a keyword search for the words that form the word node. Since the used search engine is using prefix search (see section 2.1), it is not only possible to search for full words but also for word prefixes. This is denoted by adding a *-symbol to the end of a word. For example searching for the word *play** does not only find hits for occurrences of *play*, but also for *played*, *player*, *playing* and so on.

With the combination of full-text and ontology search, it is also possible to simulate relations through the cooccurrence with words. This can be helpful for cases where the desired relation is either not contained in the ontology or when it is poorly maintained. Consider for instance the query:

Example Query 2. *"Animals that eat green plants"*

There is currently no *eats* relation for the *animal* class in our ontology, so we simulate it through the cooccurrence with the word *eat** and the class of *plants*. There is also no *has-color* relation for the *plant* class, so we just narrow it down by requiring it to co-occur with the word *green*. Supplying our tree grammar again with only these concrete terminal values we get the grammar $G_2 = \{N, T_2, S, P_2\}$, with:

$T_2 = \{\text{CLASS:animal}, \text{CLASS:plant}, \text{occurs-with}, \text{WORD:eat*}, \text{WORD:green}\}$

$$\begin{aligned} P_2: \quad CLASS &\rightarrow \text{CLASS:animal}\,[(COOCC|RELATION)^*] \\ CLASS &\rightarrow \text{CLASS:plant}\,[(COOCC|RELATION)^*] \\ ENTITY &\rightarrow \epsilon \\ COOCC &\rightarrow \text{occurs-with}\,[WORD\,(CLASS|ENTITY|WORD)^*] \\ RELATION &\rightarrow \epsilon \\ WORD &\rightarrow \text{WORD:eat*}\,[\epsilon] \\ WORD &\rightarrow \text{WORD:green}[\epsilon] \end{aligned}$$

Using grammar G_2 we can, for example, generate the tree in figure 4 to express example query number 2.

3.3. Query Results

Our user interface requires two different types of results for a query. It needs lists of proposals of the possible nodes for the query tree, so that users can build them, as well

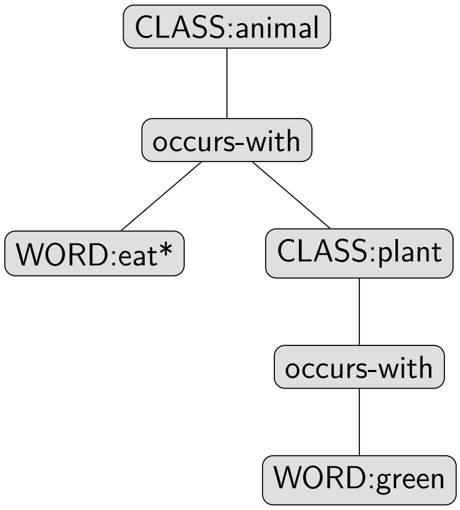


Figure 4: Sample query tree for the query “Animals that eat green plants”.

as hits to the query that are displayed to the users as results. How these two results are utilized by the user interface is explained in chapter 4.

The proposals are basically simple lists of classes, entities, relations, or words that can either be added to a selected node in the query tree or replace it. For the root node of example query number 1 such lists could look like this:

Entity proposals:

```
ENTITY:Saving_Private_Ryan
ENTITY:Empire_of_the_Sun_(film)
ENTITY:Schindler’s_List
ENTITY:1941_(film)
...
```

Relation proposals:

```
RELATION:directed-by
RELATION:created-by
...
```

Due to the combination of full-text and ontology search, the hits to our queries can consist of two different types, namely facts from the ontology and documents from the text collection. The hits are always returned for the root node of the query tree. Considering example query number 1 again, we can get the following hits with the current prototype of our search engine:

Ontology facts:

```
Saving Private Ryan is-a:  movie
Steven Spielberg directed Saving Private Ryan
Empire Of The Sun Film is a:  movie
```

Steven Spielberg directed Empire Of The Sun Film

Schindler's List is a: movie

Steven Spielberg directed Schindler's List

...

Documents:

Saving Private Ryan [2]: Saving Private Ryan is a 1998 American war film set during the invasion of Normandy in World War II

Saving Private Ryan [113]: For years now, I've been looking for the right World War II story to shoot, and when Robert Rodat wrote Saving Private Ryan, I found it

Empire of the Sun (film) [141]: Other topics that Spielberg previously dealt with, and are presented in Empire of the Sun, include a child being separated from his parents (The Sugarland Express, Close Encounters of the Third Kind, E.T. the Extra-Terrestrial, and Poltergeist) and World War II (Schindler's List, Saving Private Ryan, Close Encounters of the Third Kind, 1941, and Raiders of the Lost Ark)

Empire of the Sun (film) [6]: Spielberg was attracted to directing the film because of a personal connection to Leans films and World War II topics

List of historical drama films [570]: Schindler's List (German industrialist Oskar Schindler's assistance to keep Jewish people from being interred in concentration camps during World War II)

11572 Schindler [3]: According to the Jet Propulsion Laboratory database, the asteroid was named after Oskar Schindler, whose actions during World War II were featured in Steven Spielberg's film Schindler's List

...

As mentioned in section 2.1, the current prototype of the search engine treats each sentence of a Wikipedia article as a separate document. This is why the here listed documents consist of the name of the corresponding Wikipedia article, the number of the sentence that represents the document and the sentence itself.

4. The User Interface

This chapter presents our new semantic search user interface. It allows users to find information using semantic queries of the form that was defined in the previous chapter. Therefore, it offers a comfortable way to incrementally create such queries and displays the hits to the query in a well-arranged way.

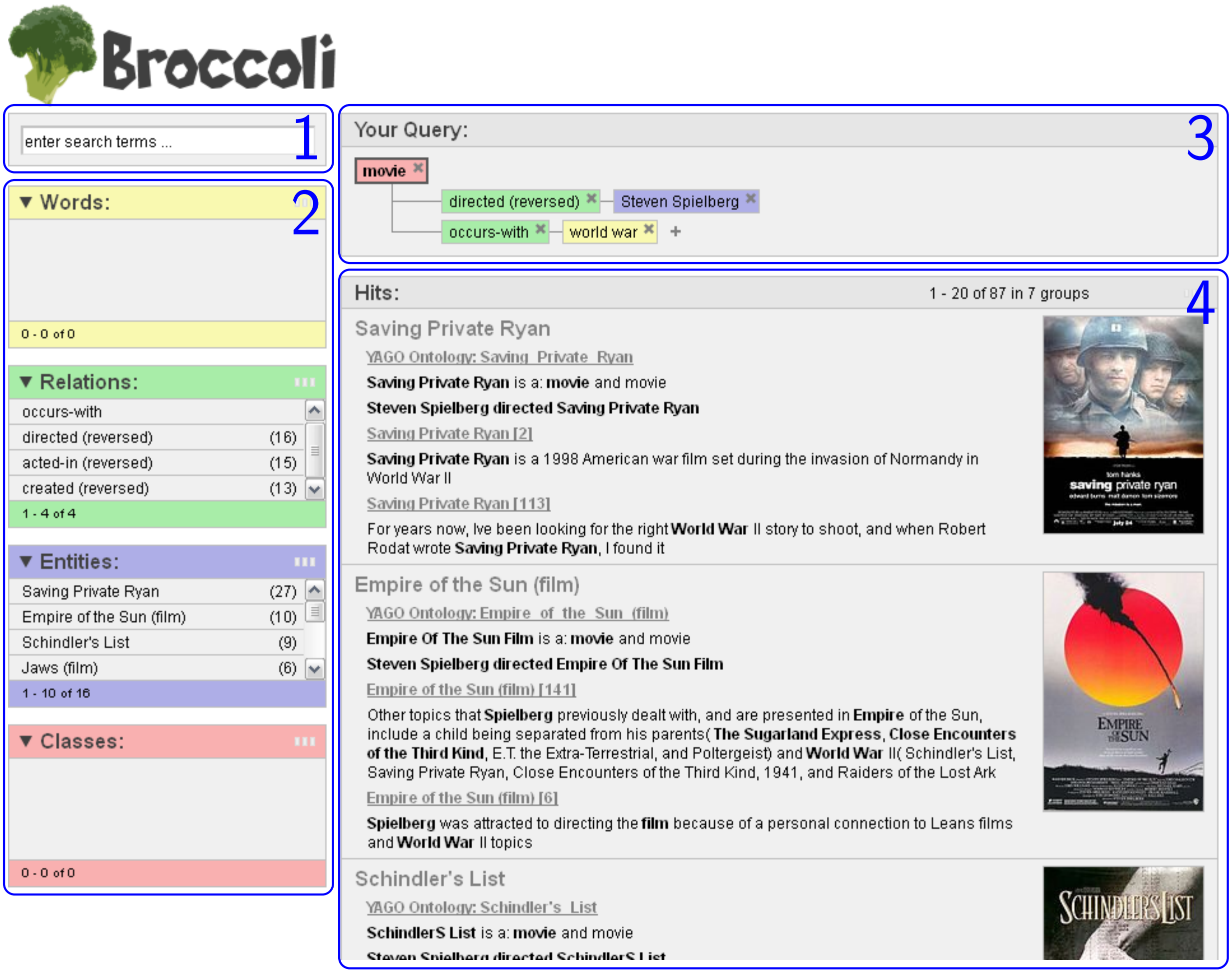
At first we show the layout of the application with the help of a screenshot. Then we explain the main elements of the user interface. We also address some advanced features that were added to further improve the usability of the application. At the end we demonstrate how the application is used with our previous query examples.

4.1. Overview

Figure 5 shows a screenshot of our user interface to give an overview of the application. The main elements of the interface are:

1. *Input Field:* The single input field of the user interface. Here the users enter what they are searching for. Upon typing, the proposal boxes automatically adapt their content to the entered prefix. Alternatively, it can also be used to add custom words directly to the query by pressing the enter key. So pressing the enter key has practically the same effect as clicking on a word proposal.
2. *Proposal Boxes:* The four proposal boxes are the key element why the user does not need any knowledge of the underlying ontology. Each proposal box has its own color that represents the type of data it provides. They show possible selection options for the user that depend on the current query and the prefix that was entered into the input field. The search query is incrementally created by selecting single proposals from these boxes. They are explained in more detail in subsection 4.2.2.
3. *Query Panel:* This is our extended breadcrumbs panel. It displays the current query as a tree as defined in section 3.2. The nodes of the tree are colored in the same manner as the proposal boxes, according to the type of information they represent. The breadcrumbs panel cannot only be used to delete certain nodes of the query tree again, but it provides more advanced query refinement features. Single nodes can be selected and can thus be replaced or further restricted by adding relations. For more details see subsection 4.2.3.
4. *Hits Area:* Here the hits to the query are displayed. Except for normal keyword search, the hits get grouped by their corresponding entities. Since the search content is currently the English Wikipedia, each group contains a link to the corresponding Wikipedia article and an image from the article, if there is one. Every hit is accompanied by an excerpt as evidence, in which all words that match the query are highlighted. See subsection 4.2.4 for more information.

Resnick and Vaughan [Resnick 06] propose some best practices specifically for search user interface design. With the presented elements of our interface and their functionality we fulfill the following of these best practices:



enter search terms ...

▼ Words:

0 - 0 of 0

▼ Relations:

occurs-with	
directed (reversed)	(16)
acted-in (reversed)	(15)
created (reversed)	(13)

1 - 4 of 4

▼ Entities:

Saving Private Ryan	(27)
Empire of the Sun (film)	(10)
Schindler's List	(9)
Jaws (film)	(6)

1 - 10 of 16

▼ Classes:

0 - 0 of 0

Your Query:

movie ✖
directed (reversed) ✖ — Steven Spielberg ✖
occurs-with ✖ — world war ✖ +

Hits: 1 - 20 of 87 in 7 groups

Saving Private Ryan

YAGO Ontology: Saving_Private_Ryan

Saving Private Ryan is a: **movie** and movie

Steven Spielberg directed Saving Private Ryan

Saving Private Ryan [2]

Saving Private Ryan is a 1998 American war film set during the invasion of Normandy in World War II

Saving Private Ryan [113]

For years now, Ive been looking for the right **World War** II story to shoot, and when Robert Rodat wrote **Saving Private Ryan**, I found it

Empire of the Sun (film)

YAGO Ontology: Empire_of_the_Sun_(film)

Empire Of The Sun Film is a: **movie** and movie

Steven Spielberg directed Empire Of The Sun Film

Empire of the Sun (film) [141]

Other topics that **Spielberg** previously dealt with, and are presented in **Empire** of the Sun, include a child being separated from his parents(**The Sugarland Express**, **Close Encounters of the Third Kind**, E.T. the Extra-Terrestrial, and Poltergeist) and **World War** II(Schindler's List, Saving Private Ryan, Close Encounters of the Third Kind, 1941, and Raiders of the Lost Ark

Empire of the Sun (film) [6]

Spielberg was attracted to directing the **film** because of a personal connection to Leans films and **World War** II topics

Schindler's List

YAGO Ontology: Schindler's_List

SchindlerS List is a: **movie** and movie

Steven Spielberg directed SchindlerS List

Figure 5: Screenshot of our user interface for example query number 1. It gives an overview of its main elements, which are the single input field **(1)**, the proposal boxes **(2)**, the query panel **(3)** and the hits area **(4)**. The input field filters the proposal boxes by typing in a keyword. The proposal boxes offer the parts to build the query tree, which is displayed in the query panel. The hits area always shows the hits for the root node of the query tree, grouped by possible entities.

- **"In the result descriptions, show the keywords in context":**
  All words that match the query are highlighted in the excerpts of the hits.
- **"Organize large sets of results into categories":**
  The hits get grouped by their corresponding entities.
- **"On the results page, provide the original query in a format that can be edited":**
  The query panel always shows the current query together with the hits and it provides possibilities to alter the query.
- **"The user interface should facilitate iterative searching by supporting the modification of queries":**
  The query panel and the proposal boxes enable iterative searching and query refinements.

4.2. Main Features

Our search user interface has some features that are crucial for its functionality. These include the previously presented main elements and the way they are working together to build a query and to display the results. This section covers these main elements of our search interface and their features in detail.

4.2.1. Interactive and Proactive

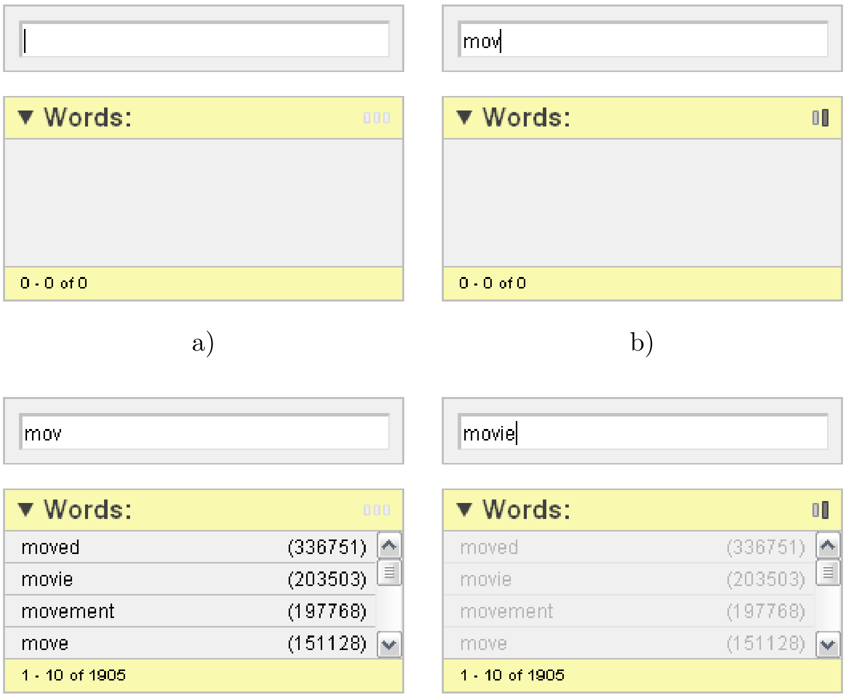
Our search user interface is a rich internet application that uses JavaScript and Ajax to provide an interactive and proactive user experience. Hence the absence of any kind of search button in figure 5. When the user enters text into the input field, the user interface automatically sends a query to the back end and displays the corresponding proposals. Figure 6 shows an example for this procedure. The hits area has nearly the same interactive behavior. It has the same loading icon in the upper right corner and as soon as the query is changed in a way that alters the hits, the new hits are loaded asynchronously and are displayed automatically. Old hits to the previous query are also grayed out while new hits are loading to denote that they do not belong to the current query.

The interactivity of our user interface is not limited to the display of proposals and hits. The process of the incremental query creation is also completely interactive and requires not a single complete reload of the page. The proposals in the proposal boxes can be clicked, what adds them to the query. Accordingly, the query tree in the query panel is automatically adapted as the query gets refined. The complete functionality of the query panel is explained in subsection 4.2.3.

By default, only a limited number of proposals and hits are displayed for each box, even if there are more available, to keep the required bandwidth low. The reloading of more is automatically done by the user interface if needed, the user does not have to press any button. When the user scrolls to the end of a proposal box or the hits, and there are more proposals or hits available, the interface creates a query for more, sends it to the back end and appends the results as soon as they are available. To signal the automatic reloading to the user, the icons in the top right corner are animated in the same way as when new proposals or hits are loaded. For the hits, there is even an additional loading bar at the bottom to indicate a reload, because the one on the top is probably not visible anymore when the page is scrolled down. In contrast to the loading of new proposals or hits, the currently displayed lists are not grayed out while a reload is in process because the displayed lists stay valid.

4.2.2. The Proposal Boxes

One goal for the user interface was that users do not need to have any knowledge about the underlying ontology. This is achieved through the four different proposal boxes. There is basically one for each kind of query tree node as defined in section 3.2, when cooccurrence is considered as a special kind of relation and values are considered to be special entities.



▼ Words:

0 - 0 of 0

a)

mov

▼ Words:

0 - 0 of 0

b)

mov

▼ Words:

moved	(336751)
movie	(203503)
movement	(197768)
move	(151128)

1 - 10 of 1905

c)

movie

▼ Words:

moved	(336751)
movie	(203503)
movement	(197768)
move	(151128)

1 - 10 of 1905

d)

Figure 6: Example for the proactive handling of user input and interactive display of proposals. Our user interface does not have a search button because input is processed automatically. Part **a)** shows the initial state of the input field and word proposal box with focus on the input field. As soon as something is typed, the interface updates itself by loading appropriate proposals. This is indicated by an animated loading icon in the upper right corner of the boxes as shown in part **b)**. When the new proposals are loaded, the animated icon is stopped and they are displayed as in part **c)**. Typing something new into the input field updates the interface again. While it is updating, already displayed proposals are grayed out to indicate that they cannot be selected anymore. This is depicted in part **d)**.

So there is one proposal box for words, one for relations, one for entities and one for classes from which the user can choose for the query creation. We assigned a different color to each of the boxes. This enables us to give the user an obvious visual cue which node of the query tree in the query display belongs to which type, because they inherit the color from the proposal box in which they appeared. Classes are currently denoted by red, entities are blue, relations are green and words are colored yellow.

The proposals that are presented in the boxes are context sensitive to the current query. This makes it easier for users to create meaningful queries that return any hits. Additionally, each proposal displays the number of hits it generates in the current context. These are the numbers in parentheses after each proposal that can be seen for example in figure 6. The input field can be used to filter the content of all proposal boxes via the entered prefix. When the user clicks on one of the proposals it is added to the query. The type of refinement to the query that is invoked by adding a proposal depends on the current query and the type of the added proposal and is explained in detail in subsection 4.2.3. Even though it causes different changes, the displayed proposals can always be selected. If a certain type of proposal cannot be added to the current query, then no proposals of that type are shown.

Our proposals are similar to facets like they are used, for instance, in Faceted Wikipedia Search. But since the proposals can be used to build complex query trees with dependencies, they provide much more possibilities than normal facets.

4.2.3. The Query Panel

The query panel is our advanced breadcrumbs display. Since our queries are trees, our interface displays the current query in this special panel instead of showing it in the input field. Showing it there would require a special syntax which a user would need to learn and which would not be as easy to read. This differs from many other search user interfaces like those from Google or Ester which use their input fields to show the query because it directly equals the input. Other search applications like Contentus or Faceted Wikipedia Search also have breadcrumb displays for their facets, but they only show the selected facets and do not offer any functionality besides the possibility to remove or deactivate single facets again. That is why we denoted our query panel as an advanced breadcrumbs display because it offers much more functionality. Figure 7 shows the query panel for a possible formulation of example query number 2. As already mentioned, the colors of the nodes are the same as the colors of the proposal boxes and denote their type, so that no prefix is needed to be able to distinguish them as in the sample trees in figures 3 and 4.

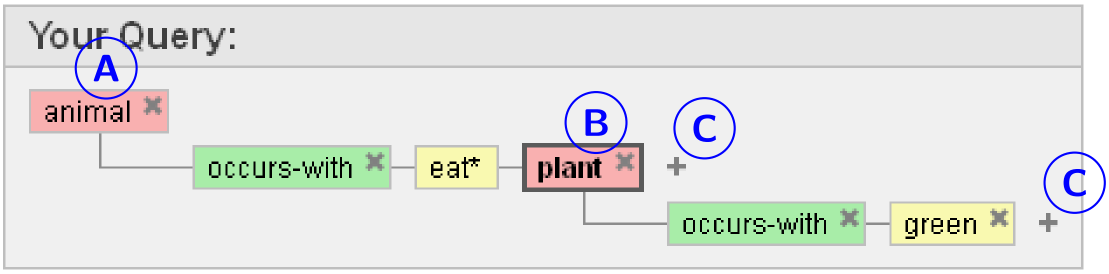


Figure 7: The query panel displaying a possible formulation of the example query "Animals that eat green plants" in our user interface. The **A** marks the root node of the query tree which is always displayed in the top left corner. The **B** marks the active node of the query tree. It can be distinguished from the other nodes by its broader border and bold font. The +-buttons that are marked by **C** belong to the cooccurrence nodes and can be used to add further co-occurring nodes.

The **A** in the figure marks the root node of the query tree. The hits in the hits area always display the results for the root node. Thus the hits area displays hits for animals for this query. If the user wants to get hits for another node of the query, he can double-click it in the query tree to re-sort it, so that this node becomes the root. The hits will then automatically update to the results of the new root node. Due to the query tree definition, this is only possible for class and entity nodes because these are the only ones that are allowed to be the root[8].

The node that is marked with **B** in figure 7 is the active node of the query tree. There is always exactly one active node in the query panel. The active node is the node on which query refinements will apply when a proposal is clicked. Accordingly, the proposal boxes always contain proposals for the active node. Table 1 lists all possible changes that are carried out when the different types of proposals are clicked, depending on the currently active node. The empty cells denote that these combinations are not possible. The active node is changed by clicking onto another node with the mouse. This node will then become active and the proposals automatically reload to adapt themselves. When a new class node is added, it is automatically set active to support a fast query generation. Note that only class, entity and word nodes can be active since the other nodes can not be refined directly.

For the cases where cooccurrence relations are added automatically (see table 1), the required word child node is always directly appended. When a cooccurrence relation is added manually by clicking onto the *occurs-with* relation in the relation proposal box, a special *WORD* placeholder node is added as the needed child node. This placeholder node must then be replaced by a real word node to activate the cooccurrence relation. Subsection 4.4.1 explains how this is done in the query creation of example query number 2.

Each node in the query tree has its own remove button on its right-hand side. When this button is clicked, the node is removed from the tree. Since the nodes depend on each other, this removes the complete sub-tree of the node. Since relations always need a target node as a child, removing the child node of a relation leads to the removal of the whole relation. The same applies to cooccurrence nodes, they always need at least one word node as a child, so they are also removed if all of their children are removed. To enable a more convenient query refinement, there exists a special feature for the removal of the source or target of a relation. A click on the remove button of such a node resets the node to the base source or target type of the relation, only a second click will remove the node completely from the tree. Subsection 4.4.2 gives an example for this refinement feature.

The +-buttons that are marked by **C** in figure 7 are a special feature of the cooccurrence nodes they belong to. They can be used to add a new child to their cooccurrence nodes. When such a button is clicked, a special *ANYTHING* placeholder is added to the cooccurrence node's children. This placeholder can then be replaced by any class, entity, or word node from the proposal boxes. An example for how this is done is contained in the query creation of example query number 2 in subsection 4.4.1.

[8] Except for word nodes, but they can only be the root for the special case of keyword only search where no other nodes can occur. For these cases, the only word node is automatically the root node and any re-sorting is not possible.

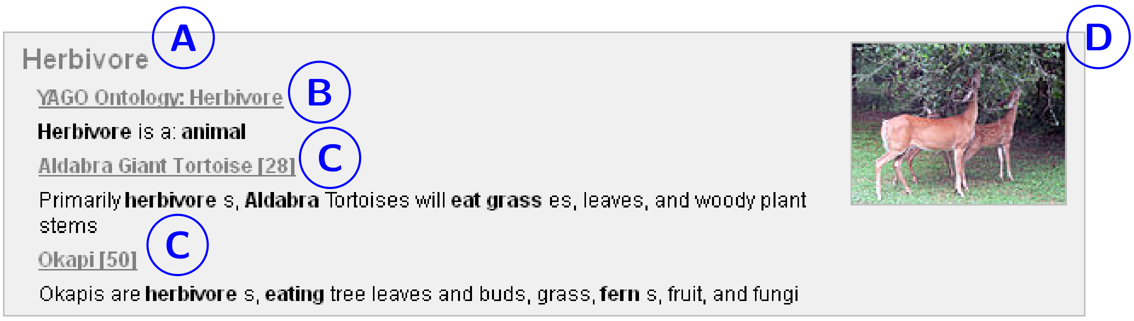
		Active Node Type			
		No active node	**Class**	**Entity**	**Word**
Selected Proposal Type	**Class**	Added as root node.	Replaces the active node.	Replaces the active node.	Replaces the active node and adds it as child with a cooccurrence relation.
	Entity	Added as root node.	Replaces the active node.	Replaces the active node.	Replaces the active node and adds it as child with a cooccurrence relation.
	Relation		Added as child to the active node together with its target class node.	Added as child to the active node together with its target class node.	
	Word	Added as root node.	Added as child to the active node with a cooccurrence relation.	Added as child to the active node with a cooccurrence relation.	Replaces the active node.

Table 1: All possible query refinements that are applied when a proposal is selected, depending on the active node in the query panel.

4.2.4. The Hits Area

The hits for the current search query are automatically displayed in the hits area of our user interface. The displayed hits always refer to the root node of the query tree. Except for normal keyword search, the hits are always grouped by entities. Figure 8 shows such a hit group of the example query number 2.

The **A** in the figure marks the name of the entity that forms the hit group. Since the current back end prototype searches in the English Wikipedia, the group name contains a link to the corresponding Wikipedia article. From subsection 3.3 we know that the hits contain ontology facts and text documents. Facts of the used YAGO ontology, in the figure marked by **B**, are always displayed first. Then there follow the document hits, marked by **C**. The current back end treats sentences of Wikipedia articles as documents. Thus a document hit consists of the name of the Wikipedia article and the sentence number as a heading that is linked to the article, and the sentence itself as an excerpt. All parts



Herbivore

YAGO Ontology: Herbivore

Herbivore is a: **animal**

Aldabra Giant Tortoise [28]

Primarily **herbivore** s, **Aldabra** Tortoises will **eat grass** es, leaves, and woody plant stems

Okapi [50]

Okapis are **herbivore** s, **eating** tree leaves and buds, grass, **fern** s, fruit, and fungi

Figure 8: One hit group for our example query "Animals that eat green plants". The **A** marks the name of the entity that forms the hit group. The hits for one group consist of ontology facts and documents. Ontology facts, marked by **B**, are always displayed first. Then there follow the document hits, marked by **C**. The parts of the excerpts that are highlighted with a bold font match parts of the query tree as evidence. The **D** marks an optional image for the group.

of the ontology facts and the document excerpts that match a node of the query tree are highlighted with a bold font, so users can directly see why these hits were returned. By default, only a few hits are displayed for every entity group (currently the ontology facts and two document hits). If the users want to see more hits for one of the entities, they can select the root as the active node in the query panel and replace it by the desired entity by choosing it from the entity proposal box. Then only hits for this entity are displayed. Finally, the **D** in the figure marks an image for the group. The image is directly obtained from the Wikipedia article that is linked with the group. When an article does not contain an image, the thumbnail is omitted.

For keyword-only search the hits are not grouped. This means that the hits panel only shows a list with the document hits. Each document hit has the same form as the document hits in the hit groups, consisting of the article name, sentence number and an excerpt. Additionally, the interface displays an image for each document hit if one is available, instead of only one image for the whole group.

Ali et al. [Ali 09] identified that visual cues, as an addition to text, are a great help for users to find results faster and to find overall better results. We have lots of visual cues in our interface: The colored query tree, hit groups, highlighted keywords and image thumbnails. The purpose of all of these is to assist users in finding the desired information as fast as possible.

4.3. Advanced Features

Besides the main features that were presented in the previous chapter, our search interface also has some advanced features that are not that obvious. However, these features contribute a great deal to the usability and comfort of the user interface.

4.3.1. Bookmark and Browser History Support

The use of JavaScript and Ajax enables websites to be interactive web applications. But in comparison to traditional websites, there are also some drawbacks. The interactivity is achieved through the modification of the DOM of a single page instead of loading completely new pages from the server. Hence the address bar of the browser does not change and does not represent the current state of the website. As a consequence, the current state of the application cannot be bookmarked because the URL always points to its initial state. Furthermore the history mechanism of a browser, which usually enables to navigate through the last visited pages with the back and forward buttons, also has no effect. These issues are also described by Holzinger et al. [Holzinger 10]. Of course there is a workaround for those problems. By setting the anchor tag of the URL via JavaScript, a web application can add state information to the URL. Thus the possibility to bookmark the current state and the functionality of the browser history can be re-enabled.

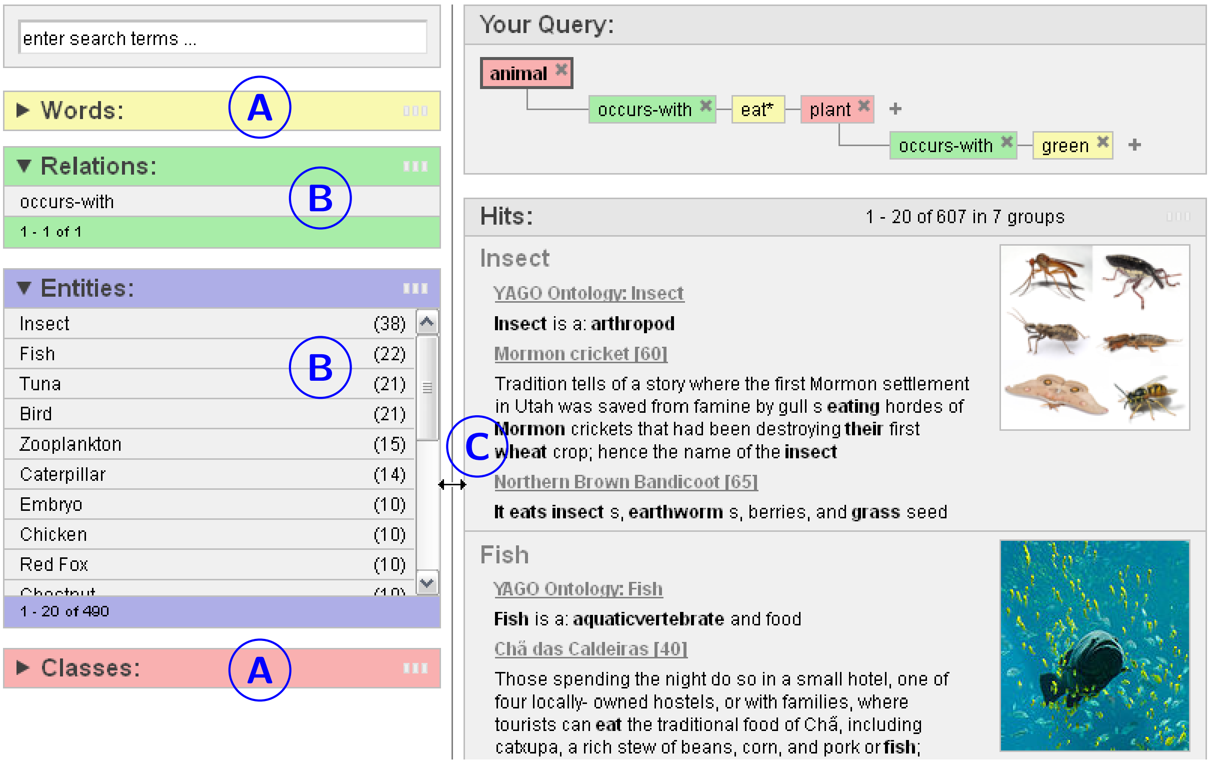
Our user interface uses the anchor tag to store its state. Therefore, users can bookmark queries or can send links of their queries to other people. The browser history is also fully functional and can be used like an undo mechanism as they are commonly known from desktop applications. The back button reverts the last change that was made to the query, while the forward button redoes it again accordingly.

4.3.2. Customizable Layout

For the convenience of our interface, its layout can be customized by adjusting the size of its components. Figure 9 shows a sample customization of the interface displaying our example query number 2.

Each proposal box can be collapsed by clicking onto its name, when it is currently not needed, to provide more space for the other proposal boxes. Another click opens the box again to its previous size. The **A** marks the collapsed proposal boxes in the figure. Additionally, each proposal box can also be independently re-sized by dragging its bottom line with the mouse to the desired size. When the mouse cursor hovers over the bottom line of a proposal box, it changes to an arrow to indicate this functionality. The **B** marks re-sized proposal boxes, the relations box was sized down and the entities box was enlarged. The last customization option is a horizontal splitter between the input field and proposal boxes on the left and the query panel and hits area on the right. By dragging it, the width of the left and right elements can be changed. When the mouse cursor is moved in between them, it changes to an arrow and a line occurs to indicate that it can be dragged. This is shown in the figure by **C**.

These customizations allow to adapt the interface to different screen resolutions and optimize the utilization of the available space. The layout settings are also stored in the URL, so they can, for instance, be saved in a bookmark and do not have to be reapplied manually every time the application is accessed.



occurs-with

1 - 1 of 1

Insect	(38)
Fish	(22)
Tuna	(21)
Bird	(21)
Zooplankton	(15)
Caterpillar	(14)
Embryo	(10)
Chicken	(10)
Red Fox	(10)
Chestnut	(10)

1 - 20 of 490

1 - 20 of 607 in 7 groups

Insect

YAGO Ontology: Insect

Insect is a: **arthropod**

Mormon cricket [60]

Tradition tells of a story where the first Mormon settlement in Utah was saved from famine by gull s **eating** hordes of **Mormon** crickets that had been destroying **their** first **wheat** crop; hence the name of the **insect**

Northern Brown Bandicoot [65]

It eats insect s, **earthworm** s, berries, and **grass** seed

Fish

YAGO Ontology: Fish

Fish is a: **aquaticvertebrate** and food

Chã das Caldeiras [40]

Those spending the night do so in a small hotel, one of four locally- owned hostels, or with families, where tourists can **eat** the traditional food of Chã, including catxupa, a rich stew of beans, corn, and pork or **fish**;

Figure 9: Layout customization example of the user interface for sample query number 2. The **A** marks collapsed proposal boxes. The **B** marks proposal boxes that were re-sized to be smaller or bigger. The draggable horizontal splitter that separates the left and right parts of the user interface is marked by **C**.

4.4. Usage Examples

The previous sections of this chapter explained how our user interface works and covered all of its features in detail. Now we show how it is used to find the desired information using our example queries. This does not only include how new queries are incrementally built, but also how queries can be refined to improve or change the results.

We claim that it is a strength of search engines like Google or our own, that the real intellectual work to formulate a query, which results in finding the desired information, is left to the users. We just try to make it as easy as possible for them. The following examples show how we try to achieve this.

4.4.1. Building Queries

This section shows how completely new queries are built. We start with our example query number 1, which was "Movies directed by Steven Spielberg that are about one of the world wars". Figure 10 shows the evolution of the example query in the incremental creation process of the user interface with screenshots of the query panel.

The query panel is initially empty as in part **a)** of the figure. We want to search for the class of movies, so we start by typing "movie" into the input field. Upon typing, the

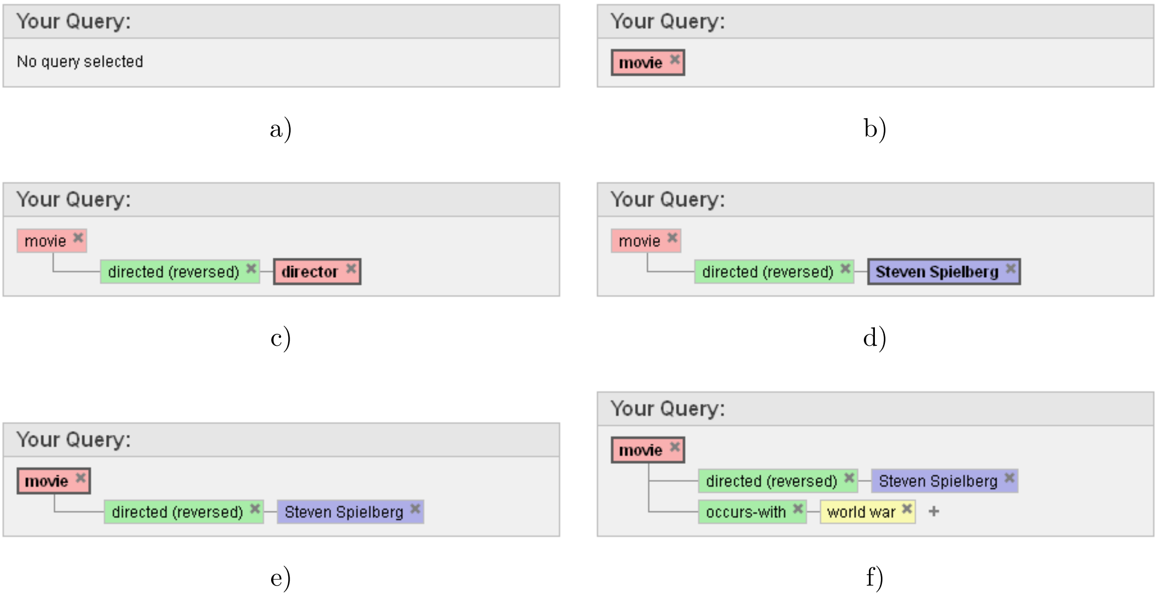


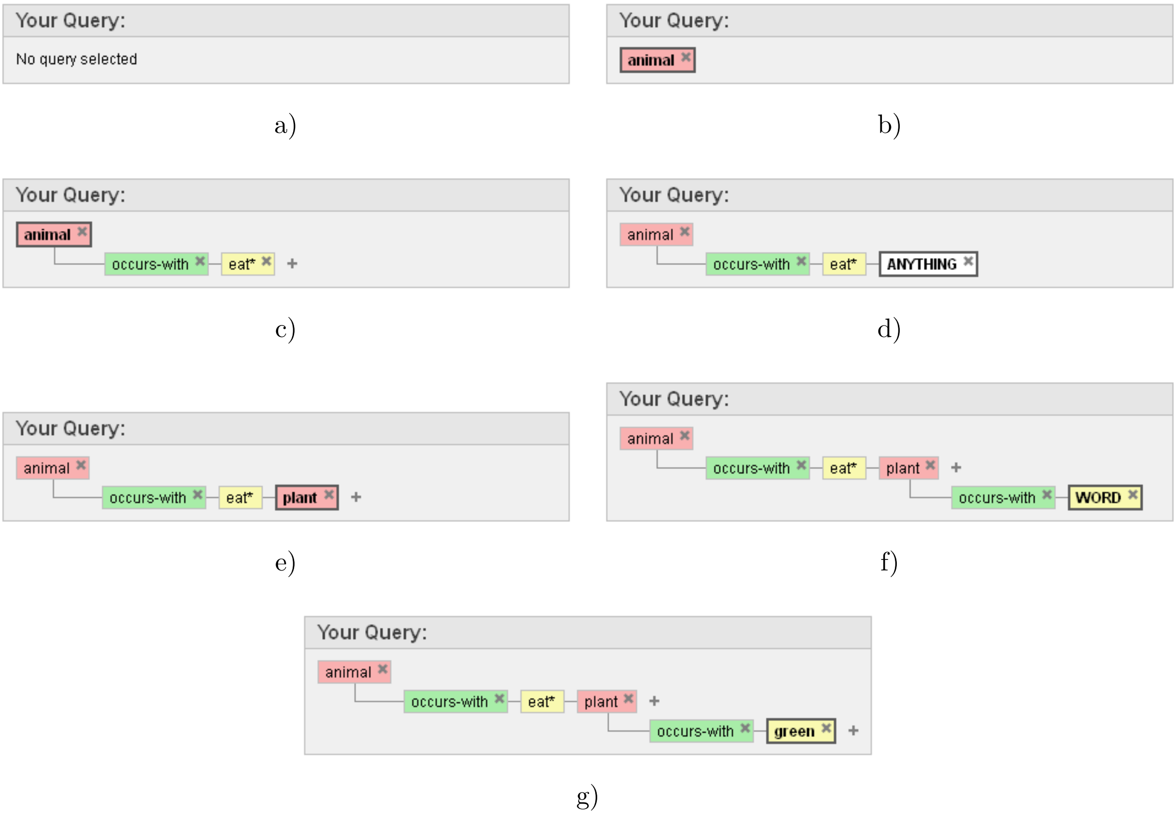
Figure 10: Possible evolution of the example query "Movies directed by Steven Spielberg that are about one of the world wars" in the incremental creation process of the user interface. The single steps are explained in detail in subsection 4.4.1.

interface automatically starts to give proposals, thus we can already see the *movie* class at the top of the class proposal box after we typed the prefix "movi". We add it to the query by clicking on it with the mouse to obtain part **b)**. The hits area now already shows movies, but we only want to get movies that were directed by Steven Spielberg. Since the *movie* class node was automatically selected as the active node, the proposal boxes already display proposals for it. So to achieve this, we can directly select the *directed(reversed)* relation[9] from the relation proposals. This appends it to the *movie* class node and also automatically adds the target class *director* and makes it the new active node as in part **c)**. Because the *director* class is now active, the entity proposals now show different directors. To search for the entity *Steven Spielberg* we can directly scroll down the proposal box until we find it, or we can at first type a prefix of "Steven" into the input field to filter it. When we found it we select it to replace the *director* class in the query, as in part **d)**. The hits area now shows movies that were directed by Steven Spielberg, but we are still missing the fact that we only want to get movies of him that are about one of the world wars. So we need to further narrow down the *movie* class by requiring a cooccurrence with the words *world war* for it. Therefore, we have to, at first, make it the active node again by clicking on it as in part **e)**. We could now choose the *occurs-with* relation from the proposals, but this is explained later in the second example. For now we just type the words *world war* into the input field and press the enter key. This automatically adds them to the *movie* class with a cooccurrence relation as in part **f)**. This completes the example query and the hits now show results for movies that were

[9] In our ontology we only have the relation "`CLASS:director RELATION:directed CLASS:movie`". To visualize it the other way round in the user interface, we display it there as "`CLASS:movie RELATION:directed(reversed) CLASS:director`". So the reversed *directed* relation basically equals the *directed by* relation.

directed by Steven Spielberg and that are about one of the world wars.

Now we show how example query number 2, which was “Animals that eat green plants”, can be built. Figure 11 shows once more the incremental construction of the query with screenshots.



a) b) c) d) e) f) g)

Figure 11: Possible evolution of the example query “Animals that eat green plants” in the incremental creation process of the user interface. The single steps are explained in detail in subsection 4.4.1.

We start again with an empty query panel as in part **a)** of the figure. We want to search for animals, so we enter “animal” or a prefix of it into the input field and choose the *animal* class from the proposals to get part **b)**. The hits now show results for all kinds of animals, but we want to find hits for animals that eat plants so we enter “eat” into the input field. Now we see that there is no relation for *eat* in the relation proposal box, so we have to express the relation with a cooccurrence. Therefore we append a *-symbol in the input field to search for the prefix “eat*” and again just press the enter key to automatically add the word node *eat** to the *animal* class with a cooccurrence relation as in part **c)**. The resulting animals should eat plants, so we need to add the class of plants to our simulated *eat*-relation. To do this we click on the +-button that is located next to the *eat** word node in the query panel. This adds an *ANYTHING* placeholder to the *occurs-with* node as in part d) and activates it. Now we can enter “plant” into the input field and select the *plant* class from the proposals to replace the *ANYTHING* node as in part e). Now we are only missing the fact that we just want green plants. Since the *plant* class node is already active, we see in the relation proposal box that there is no *has-color* relation or anything similar, so we express this with a cooccurrence again. This time we select the *occurs-with* from the relation proposals. This appends it to the *plant* node and also adds a *WORD* placeholder for the word node that is required for each cooccurrence as in part **f)**. The placeholder for the word node is already active, so we can directly enter "green" into the input field and press the enter key or select the corresponding word proposal to replace the placeholder node by the word node *green* as in part **g)**. This completes the example query. The hits now show results for animals that eat green plants.

4.4.2. Refining Queries

When searching for information, it can be very helpful to be able to search in an exploratory manner to find the query that leads to the desired results. Therefore, a search user interface must provide the possibility to refine queries. In this section we handle the refining possibilities of our interface.

Replacing class or entity nodes: Class and entity nodes in the query tree can be replaced by other classes or entities. A simple click on another class or entity proposal replaces the currently active class or entity node. Thus we can, for instance, easily search only for mammals that eat green plants, by replacing the *animal* class of example query number 2 with the *mammal* class.

Changing a word node: The content of word nodes can be changed easily. Therefore a word node has to be clicked to mark it as active. The input field is automatically filled with the current value of the word node. The new value for the word node can then be entered into the input field and a click on a word proposal or pressing the enter key changes the value of the word node accordingly. We could, for example, come up with the idea that a simulated *feeds-on* relation would suit better for example query number 2 than the *eats* relation. To change that we just have to replace the value of the *eat** word node to *feed* on*. This might lead to more accurate results.

Removing nodes: It is possible to remove sub-trees from the query with the remove button of the nodes in the query panel. Thus the query number 1 could, for instance, easily be changed to general movies about one of the world wars, by removing the *directed(reversed)* relation. For nodes that are the source or target of a relation, the remove button has the additional functionality to reset the node to the default source or target of the relation before it is removed. By using this, the query can easily be changed to search for world war movies of other directors. Removing the entity *Steven Spielberg* resets the node back to the *director* class node. When selected, it can be changed to another director through the entity proposal box, for example to *Clint Eastwood*, to only find world war movies from him.

Getting more hits for one hit group: The hit groups of the results only show a limited number of results per entity group. If more results for one entity are desired, the root of the query tree has to be replaced by this entity. Then all hits for this entity are displayed. Example query number 1 returns, for instance, Saving Private Ryan as a world war film of Steven Spielberg, but only a few document hits that concern it. By replacing the root node with the entity *Saving Private Ryan*, one can browse all document hits for it.

Expanding the query: Expanding the query is of course also a possible refinement. By appending any relations, the returned results can always be further limited. Since this is already heavily used in the query creation, we do not give another specific example here.

Permuting the query: After a user has created the example query number 2 to find hits for animals which eat green plants, he could be interested in browsing the green plants that are eaten by animals. Such query permutations are very easy with our user interface. A simple double-click onto the *plant* class node in the query panel re-sorts the query tree, so that the *plant* class node becomes the root. Since the hits are always displayed for the root, the hits area then shows the results for the query "Green plants that are eaten by animals". Figure 12 shows the resorted query tree for this query permutation.

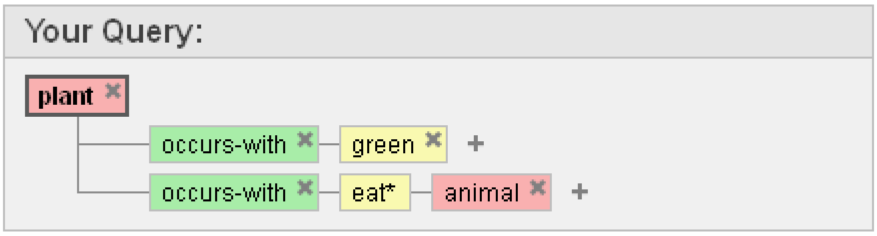


Figure 12: The query panel displaying a possible formulation of the query "Green plants that are eaten by animals", which is a permutation of example query number 2. It was created by resorting the original query tree so that the *plant* class node is the root instead of the *animal* class node.

5. Realization

This chapter covers some aspects of the realization of the user interface. At first we motivate our decision to use Google Web Toolkit as a framework. Then we explain the three-tier architecture of the interface application, before we go into detail about the implementation by describing the workflow of its classes and the difficulties that had to be resolved. At the end we show the query syntax that we use to communicate between the interface application and the search back end.

5.1. Google Web Toolkit (GWT)

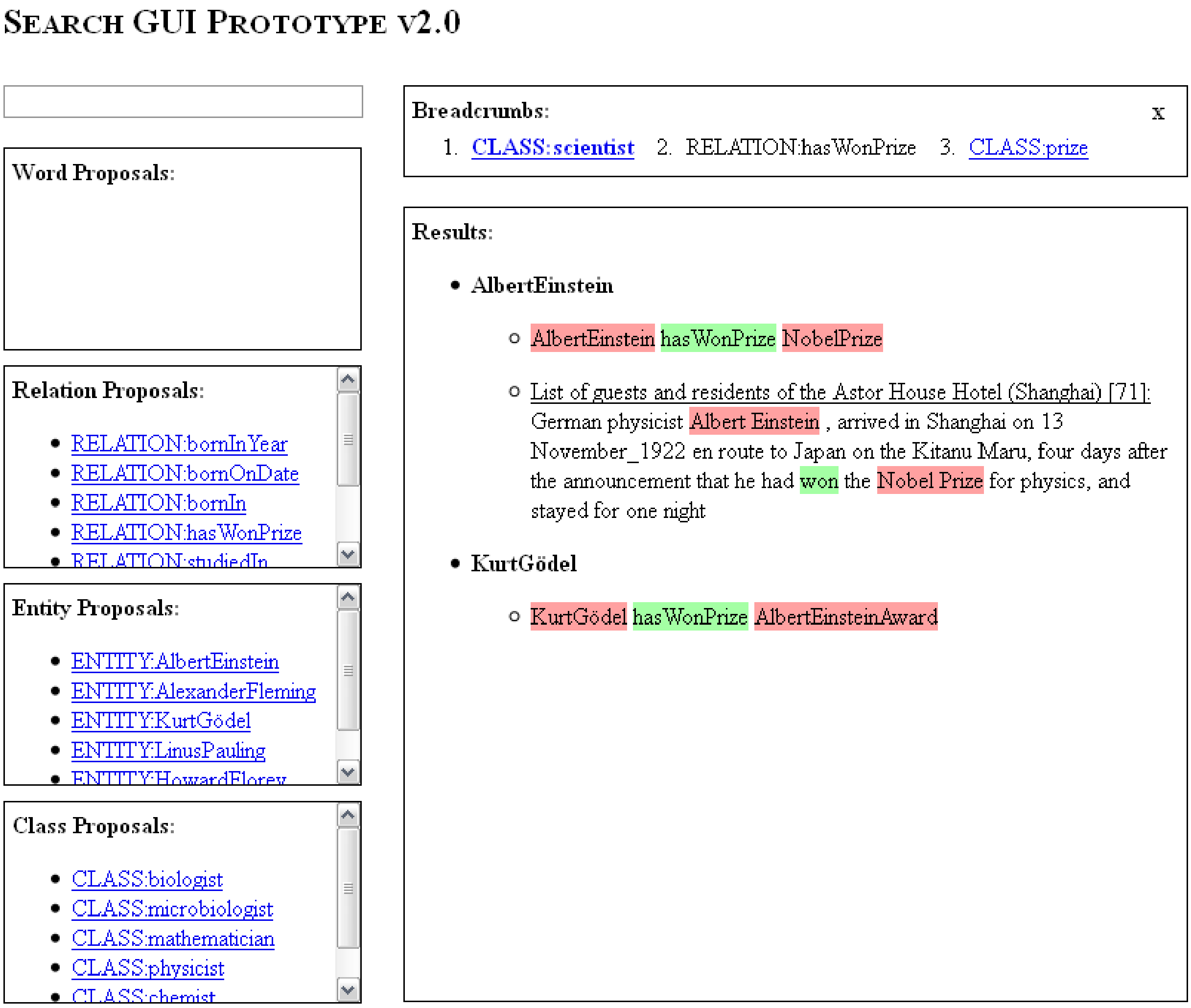
Nowadays there exist several different technologies for the development of rich internet applications. JavaScript (including Ajax) and Adobe Flash count probably to the most popular ones. Some developers are still arguing which of these technologies is the best, while some came to the conclusion that each of them has its right to exist and that they can even be combined. They all have different advantages and drawbacks and thus are suited better for some tasks and worse for others, even though it is possible to realize a wide set of different applications with all of them [Lee-Delisle 11]. We wanted our search interface to be close to normal websites and to have a similar look and feel as established web search engines like Google. So we came to the decision to use JavaScript and Ajax for our application.

Large-scale rich internet applications are usually not developed in plain JavaScript anymore. Not just the programming of low level basic functionality, like finding and manipulating HTML elements or event handling, is very tedious, the browser incompatibilities are also a big issue. There are lots of small differences between all the major browsers, their different versions, and the way they interpret certain commands or display certain elements. These have to be taken into account to support all of them. For these reasons, JavaScript frameworks and libraries, which provide comfortable APIs and cross-browser support have been developed. There are not only APIs that provide convenient abstractions for low level basic functionality, but also for advanced features like animations or even complete widgets like Date Pickers or Tab Bars. Thanks to their cross browser support, developers do not have to worry about the browser quirks anymore, because the frameworks and libraries take care of them.

One of the most popular JavaScript libraries, if not the most popular one[10], is jQuery[11]. It offers easy HTML document traversing, event handling, animation and Ajax interaction. Using jQuery, we built an early prototype for our user interface. The goal of this prototype was to get familiar with the functionality of jQuery and to be able to test if the concept of our interface is working. We began, for example, only with a single box for all proposals, but the now used basic layout crystallized already after a few iterations. Figure 13 shows a screenshot of an advanced version of the jQuery prototype that used a mock server to retrieve the displayed sample data.

[10] `http://w3techs.com/technologies/overview/javascript_library/all` and `http://trends.builtwith.com/javascript`

[11] `http://jquery.com/`



SEARCH GUI PROTOTYPE v2.0

Word Proposals:

Relation Proposals:

- RELATION:bornInYear
- RELATION:bornOnDate
- RELATION:bornIn
- RELATION:hasWonPrize
- RELATION:studiedIn

Entity Proposals:

- ENTITY:AlbertEinstein
- ENTITY:AlexanderFleming
- ENTITY:KurtGödel
- ENTITY:LinusPauling
- ENTITY:HowardFlorey

Class Proposals:

- CLASS:biologist
- CLASS:microbiologist
- CLASS:mathematician
- CLASS:physicist
- CLASS:chemist

Breadcrumbs: x

1. CLASS:scientist 2. RELATION:hasWonPrize 3. CLASS:prize

Results:

- AlbertEinstein
  - AlbertEinstein hasWonPrize NobelPrize
  - List of guests and residents of the Astor House Hotel (Shanghai) [71]: German physicist Albert Einstein , arrived in Shanghai on 13 November_1922 en route to Japan on the Kitanu Maru, four days after the announcement that he had won the Nobel Prize for physics, and stayed for one night
- KurtGödel
  - KurtGödel hasWonPrize AlbertEinsteinAward

Figure 13: Screenshot of a prototype of the user interface that was implemented with jQuery and a mock server.

With its provided functionality jQuery made it easy to rapidly create the prototype to be able to test the usability of our planned user interface. But the development also revealed the consequences of JavaScript not being a true object oriented language. The code base got quite complex pretty fast and was much more difficult to maintain than we were used to from other programming languages and their established object oriented design principles.

Google Web Toolkit[12] (GWT) is a framework for JavaScript, but it takes a different approach than most other available JavaScript frameworks. Using GWT, you do not program your application in JavaScript but in Java. The compiler of the GWT SDK translates the Java code into JavaScript that can then be deployed and run in a browser. Thus one can use the full power of established Java IDEs, including their debuggers for runtime debugging, to program JavaScript applications in an object oriented programming language. GOOGLE even provides a plugin for Eclipse that adds direct support for

[12] http://code.google.com/webtoolkit/

GWT. Besides object oriented programming, the concept of GWT has some more benefits, including code optimization at compile time, cross-browser support by generating different JavaScript files for different browsers, and a shared code base between the client and an optional server application. The separation in different files automatically speeds up the loading of the applications because only the necessary code is downloaded. Since the client side code for GWT applications is programmed in Java, all code that does not use specific GWT features can be reused on the server side of the application if it is also programmed in Java. For this case, GWT even provides a special asynchronous communication mechanism called GWT RPC[13]. It allows to directly pass Java objects back and forth between the client and the server over HTTP. Moreover the GWT SDK also offers a multitude of built in functions and an extensive user interface library that provides various different widgets. So, overall, GWT offers some nice features. Thus we also wanted to give it a try and reimplemented our prototype in GWT.

Through the development of the prototypes we learned that both, jQuery and GWT, offer by far enough functionality for our needs. And though there also exist powerful IDEs that support JavaScript development in general, including some with special jQuery support, we came to the decision to use GWT for our user interface. The key factor for our decision was the ability to program in a true object oriented language. We also need a middle-end server application that handles the communication between the user interface and the search back end. Thus another aspect in favor of GWT was the possibility of sharing code between the client and server and the convenient GWT RPC communication mechanism.

5.2. Architecture

In web development, three-tier client-server architectures are quite common. The first tier is the content that is rendered in the browser on the client side. The second tier is the middle end server application that communicates with the client and creates the content for it. And the third tier is a back end database on the server side that is accessed by the server application to create the content [Wikipedia 11]. Traditional web applications trigger a HTTP request to the web server for every change. The server then processes the request and sends a new HTML page to the client. Each request locks up the client application until the page is updated. Ajax based web applications work a bit different. They have a JavaScript based engine that runs on the client's browser. The engine intercepts user inputs, displays requested data, and handles interactions on the client side. If the engine needs more data, it asynchronously requests it from the web server, letting the user continue to interact with the application [Paulson 05].

Our user interface is realized with a similar three-tier architecture, using Ajax on the client side for the asynchronous loading of new content. Figure 14 shows our architecture and how the different layers interact with each other. The following sections explain the front end, the middle end, and the back end and their communication in more detail.

[13] Remote Procedure Call

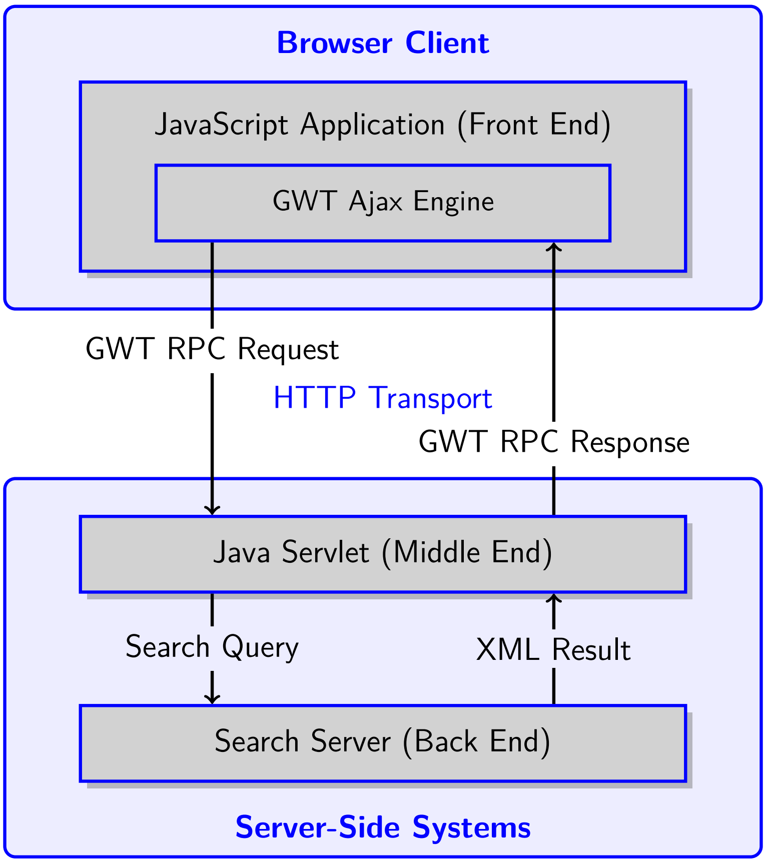


Figure 14: The three-tier client server architecture of the user interface application. The front end runs on the client side in the browser, while the middle end and back end run both on the server side.

5.2.1. Front End

The front end is the main part of the user interface application. It is the interactive JavaScript application that runs, embedded in a host page, in the client's browser. The JavaScript is generated from Java code by the GWT compiler and includes an Ajax engine from the GWT framework. This engine handles the GWT RPCs that are used by the application to communicate with the middle end for requesting new data. Using this mechanism, the front end application is able to directly call service methods of the middle end java servlet. Furthermore, Java objects can be passed to and retrieved from the server for the communication. The mechanism takes care of low-level details like the serialization of the objects for the transmission.

5.2.2. Middle End

The middle end is a Java servlet that runs on a Tomcat server and is also part of the user interface application. It uses the same data classes for the internal representation of the query and the hits as the front end application which is also programmed in Java.

Furthermore it provides remote service methods that are called by the user interface through the GWT RPC mechanism to query the results. So the middle end and front end are basically tightly coupled and their combination forms the complete user interface application. The main function of the middle end application is to act as a bridge between the front end application and the search back end server. There are several reasons why the user interface is not directly communicating with the search back end. First of all, this allows for an interface separation between them, so the JavaScript application itself is independent from the query interface of the back end. Furthermore, the current back end prototype uses HTTP to communicate, but it only implements a minimal portion of the protocol. That means that is has, for example, no support for different transfer modes or cookies and so on. Thus it is inappropriate for the direct communication over the world wide web and should only be queried over the internal network. Another reason is the Same Origin Policy[14] of JavaScript. Ajax calls are only allowed to the same host of which the page that contains the application was downloaded from, so a web server is definitely needed in-between. Besides, this allows for a more complex middle-end application that does not just forward the communication, but that further processes the queries and the results. Therefore, the client can be relieved from the parsing of queries and results to translate them into the required formats and the data that is transmitted over the web can be reduced to a minimum in an optimized format.

When our middle end application is queried from the user interface for data, it retrieves a Java object that represents a search query in the user interface application. This object is translated into a query for the back end, using the required query syntax that is shown in subsection 5.4. The search query is then sent to the back end, which returns a XML document that contains the results. The middle end parses this XML result, validates it for compliance using a XML schema file and creates a Java object from it that contains all needed information. This resulting object is then returned to the front end, which can directly work with the Java object and display its content.

In addition to serving as a bridge between the front end and the back end, the middle end fulfills another function in our application. As already mentioned, the images that are displayed together with the hits are directly obtained from their respective Wikipedia articles. Due to the Same Origin Policy of JavaScript, the JavaScript application cannot query the MediaWiki API of Wikipedia[15] itself to get these images. Thus we added a second remote service to our middle end application for this purpose. To display the images together with the results, the front end calls this service supplying the name of the desired article. The service then queries the MediaWiki API to generate the URL for a thumb with a certain maximal width for the first image that is used in the article, if it has any images at all. This URL is returned to the front end which uses it to display the image.

[14] The Same Origin Policy states that JavaScript code running on a web page may not interact with any resource not originating from the same web site. The reason this security policy exists is to prevent malicious web coders from creating pages that steal web users' information or compromise their privacy. [GoogleCode 11]

[15] http://de.wikipedia.org/w/api.php

5.2.3. Back End

The back end is the underlying search engine. As already mentioned, it is currently only a prototype. A new engine is currently in development. For this thesis, only the search content, the possible queries, and the results of the back end (see chapter 3) are important. Apart from that it is enough to know that the middle end application sends search queries to the back end (for the syntax see subsection 5.4), which returns the results in a XML document as described in the previous section.

5.3. Implementation Details

This section gives an overview of the implementation of the user interface application with GWT. It shows how the code is organized, explains the workflow of the application and mentions some difficulties that occurred during the implementation.

5.3.1. Code Organization

The user interface application, consisting of the front end JavaScript application and the middle end Java servlet, was implemented using a single GWT Java project. The code of the user interface application is structured into the following packages:

Client: The *client* package comprises the classes for the front end JavaScript application. It mostly contains classes for the different user interface components, but also some classes for debug functionality and some utility classes for advanced layout calculations.

Shared: The package *shared* contains all classes that are used by the front end client application as well as the middle end server application. It consists of all the classes that are used for the data management of the proposals, the query and the hits throughout both.

Server: The *server* package consists of the classes for the middle end Java servlet. It contains classes for its remote services, a class for the translation of the query object to a query string for the back end, and a SAX parser that creates a result object from the result XML of the back end.

5.3.2. Application Workflow

Now that we gave an overview of the organization of the code, we can give a rough insight into the workflow of the user interface application. At first, the browser of a user needs to download the host page and the embedded JavaScript application. The entry point of the JavaScript application is then the `Broccoli` class in the *client* package. It is responsible for building up the user interface in its `onModuleLoad()` method. This is done by creating instances of the classes for the different user interface elements that are located in the *client.ui* package, laying them out to each other and registering event handlers to be able to react on user input. The `Broccoli` class is in general the main controller of the front end application. It handles most of the events that are fired by the single interface elements, creates the query tree by maintaining an instance of the `Query` class of the *shared* package, initiates the GWT RPCs to the middle end and handles the asynchronous arrival of the results in the form of `BackendResult` objects from the *shared* package. Once the application is initialized, the main workflow of the application that is repeated for every change that is invoked by any user interaction is the following:

1. When the user enters something into the input field or clicks a proposal or modifies the query, the according interface elements signal this by firing corresponding events.
2. The `Broccoli` class registered to these events in the beginning, hence it gets notified of the user actions. It handles the events by adjusting its `Query` object to the change (e.g. by adding or removing a node).
3. Depending on the type of change that was invoked by the event, the `Broccoli` class decides if it is enough to update the proposals or if the hits also have to be renewed.
4. Then it notifies the user interface elements that new data is loaded and initiates an asynchronous GWT RPC to the middle end to get the new proposals for the changed query and, if necessary, a second one to get the new hits.
5. On the middle end Java servlet, the `queryBackend(...)` method of the `SearchServiceImpl` class is invoked by the GWT RPCs together with the current `Query` object, and additional information parameters that define which results should be returned.
6. The method creates an appropriate query string for the back end from the `Query` object and the desired result information using the `QueryTranslator` class.
7. This query string is sent to the back end and the XML result which it returns is thereupon parsed and validated by the `BackendXMLParser` class, which creates a `BackendResult` object from the contained information.
8. This result object is then returned to the front end application.
9. When the front end application receives the result object, it checks its content and passes the contained data to the proposal panels or the hits box respectively, so that they can update themselves to show the new data.

5.3.3. Difficulties

The only difficulties we encountered during the implementation concerned the dynamic layout of the application. The resizing capabilities for the customizable layout (see subsection 4.3.2) were actually quite tricky. The resizing was implemented so that it works independently from the CSS styles that are applied to the elements. The problem here was that the height and width that can be read from the single widget elements contain the padding of the element, but when they are set, the padding is added to the set value again, so a read and set increases the size if the element has some padding. Since GWT does not provide functionality to read single style properties like the padding or the width of the border, we had to add own native JavaScript code to be able to read such properties so that we could correctly resize the elements. Another difficulty was to get all elements to adapt to size changes of other elements. When for example the horizontal splitter between the input field and the proposal boxes on the left and the query panel and the hits area on the right is moved, and thus the dimensions of the left and right part change, the contained proposal boxes and the other elements must be notified of this change so that they can also adapt to the size change. The same holds for the query panel which resizes itself so that it is always big enough to fully contain the displayed query tree. Thus if it resizes itself, the hits area and the horizontal splitter have to adapt as well and with that also the proposal boxes and so forth. Another tricky implementation was the display of the query tree in general. To be able to draw it, we need to measure the dimensions of the single nodes, which vary depending on their arbitrary content, before we can position them in the query panel. But since elements can only be measured while they are displayed somewhere on the page, we needed to implement the `WidgetMetric` utility class to be able to measure the dimensions. This class basically temporarily adds them to a hidden panel in order to measure them.

Another difficulty that did not directly concern the layout, but that is a consequence of the customizable layout, was the saving of a user defined layout in the URL hash. When the hash is changed to store the current settings, this automatically adds a new entry in the GWT history mechanism. By default GWT does not provide the functionality to modify a history entry, but only to add new ones or to navigate through the existing ones. But we did not want new history entries for any layout changes, so that the history can only be used to undo or redo changes to the query. We are convinced that it would be quite inconvenient for users if they would have to undo several layout changes at first before they can finally undo the last query change. So we had to implement an own native JavaScript function to modify the current history entry that works on all current major browsers. Using this function, it is now possible to just add the layout information to the current URL hash without creating new history entries.

5.4. Query Syntax

This section describes the syntax of the search queries that the user interface application sends to the search back end to get the displayed results. In subsection 5.2.2 we already mentioned that the back end uses a minimal version of the HTTP protocol for the communication. So a query is sent to the back end with a HTTP GET request, where the query is encoded in the query string of the requested URL. The query string has the following syntax:

```
s=...&query=...&prefix=...
```

Additionally, a number of options can be added to define the number of certain proposals or the hits that should be returned, for example `&nofclasses=10`. The `s` parameter defines the query tree from the user interface as a list of tuples, delimited by semicolons. The `query` parameter defines which part of the query tree in `s` is searched for. The `prefix` parameter contains the content of the input field of the user interface and is thus used to filter the proposals or to get hits for keyword only search. The `s` parameter must always be existent, but it can be empty. If `s` is empty, the `query` parameter must be omitted, else it must be present. The `prefix` parameter is optional, except when `s` is empty, then it must be present and must contain a value.

Let's have a closer look on how the tuples that form the query tree for the parameter `s` are composed. Not each type of node that we defined in section 3.2 is expressed through a separate tuple, basically a tuple always defines a relation with its source and target nodes. Therefore most of the tuples are actually triples, only the cooccurrence relations are an exception because they can have more than one target and thus are defined by n-tuples with $n \geq 3$. So there is one tuple for each relation and one for each cooccurrence. The class and entity nodes of the user interface are practically just abbreviations for special relations. They are only displayed as one node in the user interface for the convenience of the users. In full length they can be expressed as "`node is-a CLASS`" and "`node equals ENTITY`" respectively. By writing them like this, the class and entity nodes can be referenced by other relations to, for instance, denote that a certain class node is restricted by another relation. That is why class and entity nodes are also defined by triples. Applying this, we get the following rules to express a query tree in tuples:

Class nodes: `$`*`number`* `is-a CLASS:`*`name`*,
e.g. `$1 is-a CLASS:movie`

Entity nodes: `$`*`number`* `equals ENTITY:`*`name`*,
e.g. `$2 equals ENTITY:Steven_Spielberg`

Relation nodes:

- Target is a class or entity node: `$`*`sourcenumber relationname`* `$`*`targetnumber`*,
  e.g. `$2 directed $1` [16]
- Target is a value node: `$`*`sourcenumber relationname`* `VALUE:`*`value`*,
  e.g. `$1 born-on-date VALUE:1889-04-20`

Cooccurrence nodes: `$`*`number`* `occurs-with` *`word`* `($`*`number`* `|` *`word`*`)*`,
e.g. `$2 occurs-with green`, or `$1 occurs-with eat* $2`

Each class and entity node is defined by its own number that must be unique in the set of triples that defines the query tree. These numbers are then referenced for relation and cooccurrence nodes. The `query` parameter in the query string must contain the number of the class or entity node which is searched for, so it must actually be the number of the root node.

This yields the following query string for example query number 1:

```
$1 is-a CLASS:movie;$2 equals ENTITY:Steven_Spielberg;$2 directed $1;$1
occurs-with world war&query=$1
```

[16] For example query number 1 we used "`CLASS:movie RELATION:directed(reversed) ENTITY:Steven_Spielberg`" in our user interface though our ontology, as stated earlier, only has the relation "`CLASS:director RELATION:directed CLASS:movie`". Reversed relations are only displayed by the interface to visualize relations the other way round. For the back end they are translated into the original relations by just swapping the source and target variables as in this example.

And this query string for example query number 2:

```
s=$1 is-a CLASS:animal;$2 is-a CLASS:plant;$2 occurs-with green;$1
occurs-with eat* $2&query=$1
```

6. Discussion

This chapter concludes the work done for our new semantic search user interface. Additionally we discuss some possible future work. This includes some features the user interface is currently missing because there was no time left to implement them, as well as features that could be nice to have to further improve its comfort and usability.

6.1. Conclusion

We have presented our new user interface for semantic full-text search. With its interactivity and proactivity, it provides a reasonably easy way for users to create semantic search queries. We showed that the queries can be displayed as trees, where the different nodes and their order form the desired semantic information. To incrementally build such a query tree, users only have to type prefixes of the classes, entities, relations or words they want to search for, to automatically get lists of proposals for possible tree nodes that can be clicked to add them to the tree. Through the help of this proposal mechanism, the users do not need to know anything about the used ontology because they can just search for available terms. Since the proposals are context sensitive to the current query, they additionally assist users in building meaningful queries that actually create any hits for the underlying search content. As hits, the user interface presents ontology facts and documents with excerpts as evidence, so that the users can easily comprehend why their query returned these results. If they did not find what they intended to get, or if they want to search for similar things, the interface provides extensive possibilities to refine their queries.

Throughout this thesis we explained everything about the user interface application. From the search content and the query language it was designed for, to the features and the usage of the interface, up to its architecture and implementation with GWT. Wherever it was practical, we used our two example queries to demonstrate certain aspects in detail. Additionally, appendix A shows some more example queries together with possible realizations with our user interface.

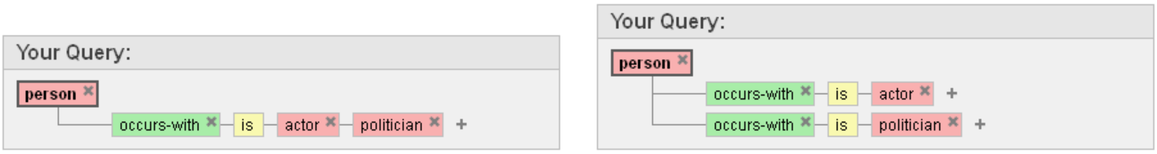
6.2. Future Work

This thesis presents the current state of the user interface application which is already fully functional and can be used to perform a multitude of different kinds of semantic search queries. Nevertheless we are aware that not all kinds of queries that one could think of are possible yet. There are still some elements missing in the query creation that have to be added in the future. Furthermore there are some additional features that proved to be valuable for the usability of other search user interfaces that also could be added to our interface. An evaluation of the user interface, for example in the form of a user study, could identify additional weak spots in our interface application that could be tweaked to improve the overall user experience. The following sections handle these aspects in more detail and give concrete examples.

6.2.1. Missing Features

Common kinds of queries can search for facts like “rivers that are longer than 50 kilometers” or “music albums that were released between 1980 and 1990”. In section 3.2 we also defined value nodes for our query language. But these values can currently only be used for simple facts like “born-on-date 1889-04-20”. It is not yet possible to define any kind of ranges for values. We will have to analyze what kinds of values and ranges are possible and then think of a good way how these could be defined in our user interface to support such kinds of queries in the future.

Another feature that our user interface does not support yet is the intersection or union of classes. Thus searching, for instance, for “people who are both actors and politicians” or “people who are either actors or politicians” is currently not directly possible. Thanks to the full-text search, these operations can be simulated again with cooccurrence as shown in figure 15, but this is not intuitive and cumbersome and might lead to lots of false positives. Therefore we need to add them as a direct feature, but we will have to test how this can be integrated nicely into our user interface.



a)

b)

Figure 15: Examples for how the intersection **a)** or union **b)** of classes could be simulated with the current user interface.

6.2.2. Additional Features

Error correction is a very handy feature of many state of the art search interfaces. When you, for example, accidentally type “*scietist*” into the GOOGLE web search, it will automatically correct your search query to “*scientist*”. This feature can not just help with typos, but also when a user does not know how to spell a word correctly, because the search engine then can propose the correct spelling and thus help the user in finding the desired results. Proposing synonyms to an entered keyword can help users in a similar way. In our interface, error correction and synonym proposals could assist users in finding the correct classes or relations or entities that they need for their queries more easily. Therefore this should be considered for future versions of the user interface.

Personalization features, like, for instance, those of CONTENTUS, could also be considered as an extension for our application. Providing the possibility to, for example, create a personal collection of queries, could have several uses. It could be valuable for users to be able to keep different queries while doing an exploratory search, so that they can easily compare them in the end. Besides this, such a collection could also be used to improve the ranking of certain proposals and hits for single users by analyzing the saved queries and thus guessing what might be more relevant for this user.

As soon as our user interface and the underlying search engine have reached a state that allows to publicize a prototype, the user interface should be extended with some help functionality. Providing some introductory sentences and clickable links to example queries like the Faceted Wikipedia Search interface could make the access to the interface much easier for people that are not yet familiar with its handling. Thanks to the interactivity of our application, it would even be possible to add an interactive tutorial mode in which the user is guided step by step through the creation of a sample query, including tooltips with descriptions for each individual step.

6.2.3. User Evaluation

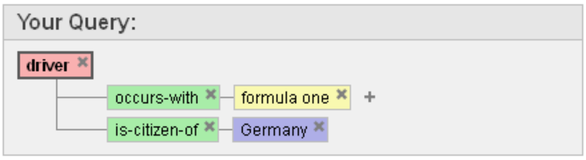
Performing user evaluations is an established method to analyze user interfaces concerning their usability. They can be of great help in finding the shortcomings of an application, both in their layout as well as their functionality, that cause problems to the users. But they also work the other way round and can make it possible to identify features that are a great advantage to the users. We did not carry out a user evaluation for our new search interface yet, but this should definitely be addressed in the future. The difficulty with user studies is that there are lots of different possible methods how they can be performed, including but not limited to automatic logging of user actions, carrying out surveys or just observing users while they are using the interface. From those, one must choose the right ones that provide the desired results. This is a problem that was analyzed by Hoeber [Hoeber 09]. They surveyed the evaluation methods that were used by numerous visual web search interfaces in recent years and propose a stepped evaluation and refinement model for the systematic study and enhancement of such interfaces. This model could, for example, be used as a base for an extensive user evaluation process.

User Evaluations can also be used to compare different user interfaces with each other. The challenge here is to evaluate them in a way that allows an objective comparison based on the obtained results. Due to the differences between most semantic search user interfaces concerning their underlying search content and their query languages and, therewith, their designs and functionality, this is a nontrivial undertaking. Hoeber and Yang [Hoeber 07] reviewed the methods that were used in such user evaluations and they provide a summary of issues that have to be taken into account when such user studies are performed. Another interesting work in this domain was presented by Wrigley et al. [Wrigley 10]. In their paper, they describe the Semantic Evaluation At Large Scale (SEALS) EU project, which aims at developing a new research infrastructure dedicated to the evaluation of semantic web technologies. The SEALS project provides a systematic methodology to test semantic search tools both automatically and interactively with human users. This allows them to test functional performance measures like precision and recall, as well as usability issues, such as ease of use and comprehensibility of the query language. So when it comes to comparing our user interface to other semantic search user interfaces, the SEALS project should be given a closer look.

A. Appendix: More Example Queries

Here we list some more queries and how they could be formulated with our user interface, to provide more examples for possible queries:

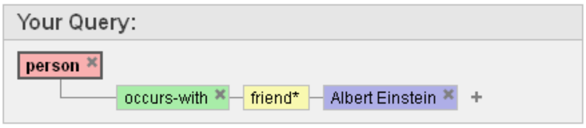
- German formula one drivers:



Your Query:

driver ✖
occurs-with ✖ — formula one ✖ +
is-citizen-of ✖ — Germany ✖

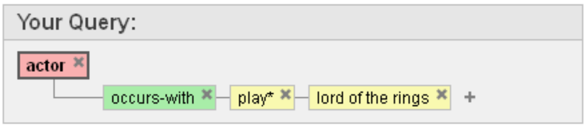
- Friends of Albert Einstein:



Your Query:

person ✖
occurs-with ✖ — friend* — Albert Einstein ✖ +

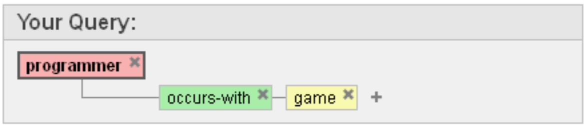
- Actors that played in The Lord of the Rings:



Your Query:

actor ✖
occurs-with ✖ — play* ✖ — lord of the rings ✖ +

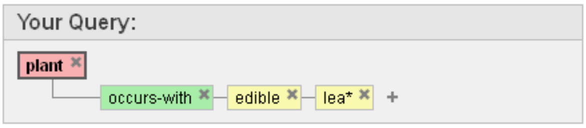
- Game programmers:



Your Query:

programmer ✖
occurs-with ✖ — game ✖ +

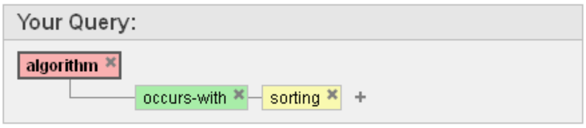
- Plants with edible leaves:



Your Query:

plant ✖
occurs-with ✖ — edible ✖ — lea* ✖ +

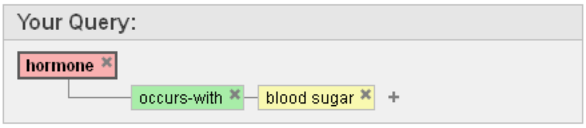
- Sorting algorithms:



Your Query:

algorithm ×
occurs-with × sorting × +

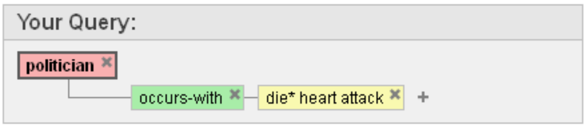
- Hormones that play a role for blood sugar:



Your Query:

hormone ×
occurs-with × blood sugar × +

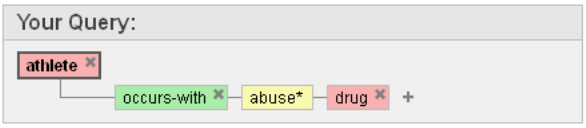
- Politicians that died of a heart attack:



Your Query:

politician ×
occurs-with × die* heart attack × +

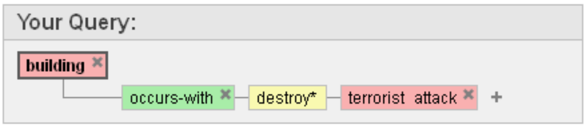
- Athletes with known drug abuse:



Your Query:

athlete ×
occurs-with × abuse* drug × +

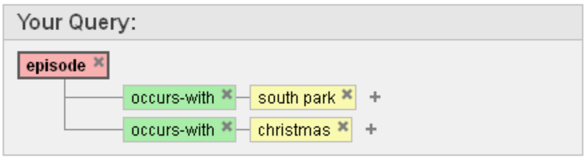
- Buildings that were destroyed in terrorist attacks:



Your Query:

building ×
occurs-with × destroy* terrorist attack × +

- South Park Christmas episodes:



Your Query:

episode ×
occurs-with × south park × +
occurs-with × christmas × +

List of Figures

Bibliography

[Ali 09] Hilal Ali, Falk Scholer, James A. Thom & Mingfang Wu. *User interaction with novel web search interfaces.* In Proceedings of the 21st Annual Conference of the Australian Computer-Human Interaction Special Interest Group: Design: Open 24/7, OZCHI '09, pages 301–304, New York, NY, USA, 2009. ACM.

[Bast 07] Holger Bast, Alexandru Chitea, Fabian Suchanek & Ingmar Weber. *ESTER: efficient search on text, entities, and relations.* In Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR '07, pages 671–678, New York, NY, USA, 2007. ACM.

[Broekstra 03] Jeen Broekstra, Arjohn Kampman & Frank van Harmelen. *Sesame: An Architecture for Storin gand Querying RDF Data and Schema Information.* In Spinning the Semantic Web, pages 197–222, 2003.

[Buchhold 10] Björn Buchhold. Susi: Wikipedia search using semantic index annotations. Master's thesis, University of Freiburg, 2010.

[CO-ODE 11] CO-ODE. *The Manchester OWL Syntax.* `http://www.co-ode.org/resources/reference/manchester_syntax/`, June 2011.

[Corby 06] O. Corby, R. Dieng-Kuntz, F. Gandon & C. Faron-Zucker. *Searching the semantic Web: approximate query processing based on ontologies.* Intelligent Systems, IEEE, vol. 21, no. 1, pages 20 – 27, jan.-feb. 2006.

[DBpedia 11] DBpedia. *About.* `http://wiki.dbpedia.org/About`, June 2011.

[Di Martino 10] Beniamino Di Martino. *An Approach to Semantic Information Retrieval Based on Natural Language Query Understanding.* In Florian Daniel & Federico Facca, editeurs, Current Trends in Web Engineering, volume 6385 of *Lecture Notes in Computer Science*, pages 211–222. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16985-4_19.

[DSpace 11] DSpace. *End User Faq.* `https://wiki.duraspace.org/display/DSPACE/EndUserFaq`, June 2011.

[Gao 11] Mingxia Gao, Jiming Liu, Ning Zhong, Furong Chen & Chunnian Liu. *SEMANTIC MAPPING FROM NATURAL LANGUAGE QUESTIONS TO OWL QUERIES.* Computational Intelligence, vol. 27, no. 2, pages 280–314, 2011.

[GoogleCode 11] GoogleCode. *Communicate with a Server - FAQ - What is the Same Origin Policy, and how does it affect GWT?* `http://code.google.com/intl/de-DE/webtoolkit/doc/latest/FAQ_Server.html#What_is_the_Same_Origin_Policy,_and_how_does_it_affect_GWT?`, June 2011.

[Hahn 10] Rasmus Hahn, Christian Bizer, Christopher Sahnwaldt, Christian Herta, Scott Robinson, Michaela Bürgle, Holger Düwiger & Ulrich Scheel. *Faceted Wikipedia Search.* In Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, Witold Abramowicz & Robert Tolksdorf, editeurs, Business Information Systems, volume 47 of *Lecture Notes in Business Information Processing*, pages 1–11. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-12814-1_1.

[Hildebrand 06] Michiel Hildebrand, Jacco van Ossenbruggen & Lynda Hardman. */facet: A Browser for Heterogeneous Semantic Web Repositories.* In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold & Lora Aroyo, editeurs, The Semantic Web - ISWC 2006, volume 4273 of *Lecture Notes in Computer Science*, pages 272–285. Springer Berlin / Heidelberg, 2006. 10.1007/11926078_20.

[Hildebrand 07] Michiel Hildebrand, Jacco van Ossenbruggen & Lynda Hardman. *An Analysis of Search-based User Interaction on the Semantic Web.* Rapport technique E0706, CWI. Information Systems [INS], 2007.

[Hoeber 07] O. Hoeber & Xue Dong Yang. *User-Oriented Evaluation Methods for Interactive Web Search Interfaces.* In Web Intelligence and Intelligent Agent Technology Workshops, 2007 IEEE/WIC/ACM International Conferences on, pages 239 –243, nov. 2007.

[Hoeber 09] O. Hoeber. *User Evaluation Methods for Visual Web Search Interfaces.* In Information Visualisation, 2009 13th International Conference, pages 139 –145, july 2009.

[Holzinger 10] A. Holzinger, S. Mayr, W. Slany & M. Debevc. *The influence of AJAX on Web usability.* In e-Business (ICE-B), Proceedings of the 2010 International Conference on, pages 1 –4, july 2010.

[Koutsomitropoulos 11] Dimitrios Koutsomitropoulos, Ricardo Borillo Domenech & Georgia Solomou. *A Structured Semantic Query Interface for Reasoning-Based Search and Retrieval.* In Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer & Jeff Pan, editeurs, The Semantic Web: Research and Applications, volume 6643 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21034-1_2.

[Lee-Delisle 11] Seb Lee-Delisle. *HTML5 vs Flash - the aftermath.* `http://sebleedelisle.com/2011/01/html5-vs-flash-the-aftermath/`, June 2011.

[Lei 06] Yuangui Lei, Victoria Uren & Enrico Motta. *SemSearch: A Search Engine for the Semantic Web.* In Steffen Staab & Vojtech

Svátek, editeurs, Managing Knowledge in a World of Networks, volume 4248 of *Lecture Notes in Computer Science*, pages 238–245. Springer Berlin / Heidelberg, 2006. 10.1007/11891451_22.

[Mani 03] Murali Mani & Dongwon Lee. *XML to Relational Conversion Using Theory of Regular Tree Grammars.* In Stéphane Bressan, Mong Lee, Akmal Chaudhri, Jeffrey Yu & Zoé Lacroix, editeurs, Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web, volume 2590 of *Lecture Notes in Computer Science*, pages 81–103. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36556-7_6.

[MPII 11] MPII. *YAGO.* http://www.mpi-inf.mpg.de/yago-naga/, June 2011.

[Neumann 08] Thomas Neumann & Gerhard Weikum. *RDF-3X: a RISC-style engine for RDF.* Proc. VLDB Endow., vol. 1, pages 647–659, August 2008.

[Paulson 05] L.D. Paulson. *Building rich web applications with Ajax.* Computer, vol. 38, no. 10, pages 14 – 17, oct. 2005.

[Resnick 06] Marc L. Resnick & Misha W. Vaughan. *Best practices and future visions for search user interfaces.* Journal of the American Society for Information Science and Technology, vol. 57, no. 6, pages 781–787, 2006.

[W3C 11a] W3C. *OWL 2 Web Ontology Language.* http://www.w3.org/TR/owl2-overview/, June 2011.

[W3C 11b] W3C. *Resource Description Framework (RDF).* http://www.w3.org/RDF/, June 2011.

[W3C 11c] W3C. *SPARQL Query Language for RDF.* http://www.w3.org/TR/rdf-sparql-query/, June 2011.

[Waitelonis 11] Jörg Waitelonis, Johannes P. Osterhoff & Harald Sack. *More than the Sum of its Parts: CONTENTUS – A Semantic Multimodal Search User Interface.* In Proceedings of Workshop on Visual Interfaces to the Social and Semantic Web (VISSW), Co-located with ACM IUI 2011, Feb 13, 2011, Palo Alto, US, CEUR Workshop Proceedings, volume 694, 2011.

[Wikipedia 11] Wikipedia. *Multitier architecture.* http://en.wikipedia.org/wiki/Multitier_architecture, June 2011.

[Wrigley 10] Stuart N. Wrigley, Dorothee Reinhard, Khadija Elbedweihy, Abraham Bernstein & Fabio Ciravegna. *Methodology and campaign design for the evaluation of semantic search tools.* In Proceedings of the 3rd International Semantic Search Workshop, SEMSEARCH '10, pages 10:1–10:10, New York, NY, USA, 2010. ACM.

[Zhou 07] Qi Zhou, Chong Wang, Miao Xiong, Haofen Wang & Yong Yu. *SPARK: Adapting Keyword Query to Semantic Search.* In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber & Philippe Cudré-Mauroux, editeurs, The Semantic Web, volume 4825 of *Lecture Notes in Computer Science*, pages 694–707. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-76298-0_50.