NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Mats Jørgen Skaslien

Lister: A Hybrid Approach for User-friendly Semantic Web Information Retrieval

Masteroppgave i Datateknologi
Veileder: Trond Aalberg
Juni 2020

Mats Jørgen Skaslien

Lister: A Hybrid Approach for User-friendly Semantic Web Information Retrieval

Masteroppgave i Datateknologi
Veileder: Trond Aalberg
Juni 2020

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Abstract

Semantic Web search engines can give access to vast amounts of structured data, but the general public struggle when operating them. Studies in the field indicate that poor *usability* may be the root issue. The overarching goal of this thesis is, therefore, to open the semantic web to casual users by exploring the implementation of a more user-friendly search engine.

By conducting a literature review, key usability challenges for Semantic Web search engines were identified. Based on findings from the review, a *hybrid* search engine using natural language query formulation and interactive query specification was proposed. A user-driven development process was conducted to further ensure end product usability, resulting in the *Lister*[1] search engine. Lister was used to examine whether such an approach could improve usability and aid users in data exploration.

User tests were held both during and after the development process providing qualitative data on the usability of the system. As a result, several usability issues were uncovered and subsequently improved upon. Data from user interviews also indicated that casual users operated Lister with ease. Lastly, 86 participants completed the System Usability Scale test, forming a quantitative measure of usability. SUS scores show that Lister outperforms similar tools reviewed in this report in terms of usability, as well as the overall average for web user interfaces.

The results suggest that user-driven development is a valuable tool in creating more usable Semantic Web Search engines. Qualitative data also indicate that users respond well to the hybrid query-approach put forth by Lister. These findings are a step forward with regards to usability in Semantic Web search engines. Using Lister, the general public can draw advantage of the power that the web of data provides in a user-friendly way.

[1]Available online at `https://folk.ntnu.no/matsjsk/`

Sammendrag

Søkemotorer for det Semantiske Nettet kan gi tilgang til store mengder strukturerte data, men allmennheten sliter med å ta dem i bruk. Studier på området indikerer at dårlig *brukbarhet* kan være roten av problemet. Det overordnede målet med denne oppgaven er derfor å åpne det Semantiske Nettet for et bredt publikum ved å utforske implementasjonen av en mer brukbar søkemotor.

Viktige brukervennlighetutfordringer som søkemotorer for det Semantisk Nettet står overfor ble identifisert ved å gjennomføre en litteraturgjennomgang. Basert på data fra gjennomgangen ble en *hybrid* søkemotor som tar i bruk naturlig språk grensesnitt og interaktiv spørringsspesifikasjon foreslått. En brukerstyrt utviklingsprosess ble utført for ytterligere å sikre sluttproduktets brukbarhet, noe som resulterte i søkemotoren *Lister* [2]. Lister ble brukt til å undersøke om en slik tilnærming kunne gi forbedret brukervennlighet og hjelpe brukere i utforskning av data.

Kvalitative data om brukbarheten til systemet ble gjennom brukertester innhentet både under og etter utviklingsprosessen. Som et resultat ble flere brukbarhetsproblemer avdekket og deretter forbedret, og data fra brukerintervjuer indikerte at brukere enkelt dro nytte Lister. Til slutt fullførte 86 deltakere System Usability Scale-testen og dannet et kvantitativt mål på systemets brukervennlighet. SUS-scoren viser at Lister overgår sammenlignbare verktøy som er gjennomgått i denne rapporten når det gjelder brukervennlighet, så vel som det totale gjennomsnittet for nettbrukergrensesnitt.

Resultatene antyder at brukerstyrt utvikling er et verdifullt verktøy i å lage mer brukbare søkemotorer for det Semantiske Nettet. Kvalitative data indikerer også at brukere reagerer godt på den hybride spørringstilnærmingen Lister har fremmet. Disse funnene er et skritt fremover med hensyn til brukbarhet i semantiske søkemotorer, og viser at ved å bruke Lister kan allmennheten utnytte kraften som det Semantiske Nettet gir på en brukervennlig måte.

[2]Tilgjengelig online på `https://folk.ntnu.no/matsjsk/`

Acknowledgements

I would like to thank my supervisor, Trond Aalberg, at the Department of Computer Science at NTNU, for his help. Trond provided me with countless hours of guidance and meticulously revised this report. He was a great sparring partner when it came to developing the ideas in this paper, and I have learned a lot from him. I am incredibly grateful for his help, which has been invaluable for the project.

I would also like to thank my parents, as well as Marit Johanne and Fredrik Skatvedt for opening their homes for me during the Covid-19 pandemic. Due to the pandemic, I cut my stay in Austria short, leaving me without a suitable working space. Being allowed to move in and use their homes as offices helped my work tremendously, for which I am grateful.

Finally, I would like to extend a thank you to all participants of the various stages of user testing. Without you, this project wouldn't have been feasible in its current form. I was utterly overwhelmed by the quality of feedback and willingness to help and participate that was displayed by complete strangers. I hope this project can be of some use to you.

Contents

Figures

Tables

Chapter 1

Introduction

This chapter provides an overview of the motivation behind the project, what methods were used, and an overview of the report structure.

1.1 Motivation

The Semantic Web is also known as the Web of Data. This data web is vast, free, and open, but is mainly accessed through the structured query language SPARQL. This sets a high threshold for the wider public to gain access and draw advantage of the knowledge and information contained therein.

The structured nature of the data is also the greatest feature of the Semantic Web. The structure allows a Semantic Web user to retrieve data based on *their* exact need; The user can get for example a list of all Norwegian mountains between 1500 and 2000 meters, or all airports with a name beginning with 'X', or something more niche[1] still.

Several user interfaces have been made to open the Semantic Web to casual users. They aim to simplify access by hiding the complexities of query languages and data fetching from the user, handling querying in the background. These interfaces are however complex and poorly designed, and users struggle when operating them [1–4]. The goal of this thesis is, therefore, to build a more user-friendly Semantic Web search engine, so that the general public may draw advantage of the Semantic Web.

[1]Or general, such as retrieving all persons on Wikipedia.

1.2 Research Questions and Hypotheses

1.2.1 Research Questions

The overarching goal of creating an easy-to-use Semantic Web search engine was broken down into several research questions. These questions specify the focus areas of the project, and are accompanied by a small text describing how they relate to the research goal.

Research Goal: Simplify the search for- and retrieval of information from databases of Semantic Web data for casual users.

R.Q. 1 *Can interactive techniques and full-text search help users in query formulation?*
Query formulation is the starting point of the searching process; it is therefore crucial that the user gets a 'smooth' start.

R.Q. 2 *Can interactive techniques help users in data exploration?*
Whether through filtering, ranking, or adding or removing information, interaction helps in understanding a dataset better. By integrating such features into the search engine itself instead of relying on third-party software like Excel, the exploration of the data might be simplified.

R.Q. 3 *Can data topology be exploited in result ranking?*
In large databases there may be many items that are *lexically* similar to a query, yet are unrelated or uninteresting for the user. It is therefore of interest to see whether the topology of semantic data can be exploited to improve upon the initial term-based result ranking.

R.Q. 4 *Can data topology be exploited in explorative search?*
Knowing the properties and relationships of a given dataset might aid the user in understanding the context, aiding them in asking better 'questions' of their data.

R.Q. 5 *Can user-driven development lead to more user-friendly interfaces?*
The design decisions made by a developer may not align with the needs and desires of the end-user. Involving potential end-users in the design and development process might, therefore, ensure that the target audience finds the finished product easy to use.

1.2.2 Hypotheses

To further focus and formalize the research questions, each one was restated into a hypothesis. By reducing the problem-space of a research question to a testable hypothesis, it is also easier to test the outcome of the study.

H. 1 Full-text search, continuous feedback, and iterative query specification will ensure that query formulation is easy.

H. 2 By displaying data in a fully interactive data-table, users may explore their data at a glance. They may also drill down by filtering, ranking, or following a *URI*.

H. 3 Query result ranking can be improved by scoring results based on the *degree*[2] of the retrieved results, and pruning irrelevant entities.

H. 4 Data exploration will be simplified by exploiting data topology to fetch relevant information about the chosen subject, and add human-readable labels to entity URI's[3]

H. 5 By performing iterative development with continuous user testing, a better than average SUS result can be achieved.

[2]The degree of a given entity is the number of other entities it is linked to. For a class entity, it is the number of members that class has.

[3]Unique Resource Identifiers are used to define individual entities on the Semantic Web. These URI's are hard for humans to read.

1.3 Method

This section provides an overview of the research and development methodology used in this thesis. The work was structured into three phases;

The Research Phase A literature search was performed, and a review of surveys on the usability of Semantic Web search engines was conducted. Several issues regarding the usability of Semantic Web search tools were identified from the information gathered. A novel semantic search engine was proposed to address the usability challenges, along with a workflow and testing framework for ensuring a usable end-result. The search engine combines natural language query formulation with interactive table representations for semantic data.

The Development Phase The development phase implemented the specification made in the first phase. An iterative design and development process was completed, performing user-testing at two critical stages underway. Several usability issues were therefore discovered early in the development and addressed. The finished project was deployed online[4], open and available for all interested parties.

The Evaluation Phase Quotes, behaviour, and feedback from users were compiled from the user-tests. This data formed the basis for the qualitative evaluation of the usability of the Lister search engine and was used to test the validity of the hypotheses. The System Usability Scale test was used as a quantitative measure of usability. Potential users were invited to test the system and rate it using the SUS scale. A total of 86 participants completed the test, providing data for evaluating the usability of the system. Users were also encouraged to leave comments and feedback on their experience using Lister, which provided qualitative feedback on the completed system. The research questions and hypotheses were answered using the collected data, concluding the project.

[4]Available now at `https://folk.ntnu.no/matsjsk/`

1.4 Thesis Outline

Chapter 1 - Introduction has introduced this thesis, presenting the motivation and structure of the work, and an overview of the report.

Chapter 2 - Background presents the theoretical background needed for readers not familiar with the Semantic Web, reviews similar work, and forms the theoretical foundation for the rest of the thesis.

Chapter 3 - Method details how the project was planned, what considerations were made, how work proceeded and what methods were chosen to evaluate the results of the project.

Chapter 4 - Development presents the reader with how the Lister search engine is used, the high-level architecture of the system, as well as implementation details and what technologies were chosen.

Chapter 5 - Results and Discussion presents the results gathered during testing. These results are compared to similar projects' results, followed by a discussion about the project and its' results as a whole.

Chapter 6 - Conclusion concludes the thesis, summing up the work done and the contributions made. It also outlines areas for future work.

Chapter 2

Background

2.1 What Is the Semantic Web?

The World Wide Web revolves around *documents* known as ‘websites’ which are interconnected using *hyperlinks*, more colloquially known as *links*. The Semantic Web - often referred to as the Web of Data - is an extension of that Web, made for *computers*.

On the Semantic Web, there are *resources* instead of documents, and *statements* instead of hyperlinks. This Web is called semantic, as the statements *say something* about the resources they connect. The Semantic Web of resources and statements is made of ‘clean’ machine-readable data. This is in stark opposition to the traditional Web, where websites have complex structures and hard-to-parse content made to be pleasing to the human eye.

A Semantic Web resource represents a physical or digital *thing*, or an abstract concept. Resources have a location and a name, both of which are defined by their Uniform Resource Identifier (URI). An example URI for the city Trondheim is `http://dbpedia.org/page/Trondheim`. URIs might look familiar, as URLs - or links - are a subclass of URIs. The example URI uniquely names Trondheim as an entity, and that its’ data can be obtained by requesting the data stored at `http://(...)`.

Statements can be made about these resources, stating what we know about the entity. We know, for example, that Trondheim lies in Norway, and we can state this and any other fact using a *triple*. A triple consists of a *subject*, a *predicate*, and an *object*. In this case, we can say that the subject `Trondheim` has the predicate `Country` with the object value `Norway`. This method of stating facts about URIs is part of the Resource Description Framework or RDF for short.

RDF defines no schema for storing *things*. It is never stated that a city such as Trondheim *must* be located in a `Country`, even though most are. Each fact lives on its own, linking and identifying its little part of the Semantic Web. This makes it easy to add entities and information to the Semantic Web; There is no need to know all the information about a thing, and no need to follow a universal schema designed for that thing.

Getting information from the Semantic Web is, typically, done using the structured query language SPARQL. A Query is written by describing what the desired data 'looks like'. To retrieve a list of Norwegian cities, a query asking for *things* that are cities and further stating that those *things* are located in Norway would do the trick. That query, written in a simplified pseudo-SPARQL would look like this;

```
SELECT ?City {
  ?City isA City.
  ?City Country Norway.
}
```

Question marks (?) in SPARQL indicate *variables*, and words without a leading question mark are *URI*s or predicates[1]. The statement `SELECT ?City` says that values of the `?City` variable should be returned. Finally, to mark the end of a statement, a period (.) is used. A *knowledgable* user can string together statements like these, creating arbitrarily complex queries.

Queries are made to a triple-store, which contains the data of the Semantic Web. Some triple-stores are topic-specific, such as MusicBrainz[2] which stores large amounts of music metadata. Others, like DBpedia[3] and Wikidata[4], span a number of domains, aiming to create a general *knowledge-base*. Wikidata is a Wikimedia project that stores structured data from other Wikimedia projects, such as Wikipedia, Wikivoyage, Wiktionary, and Wikisource. Much of the information on Wikidata is automatically imported. However, as its' sibling Wikipedia, it also has a large community of users curating its data. Due to its active community and large amounts of high-quality data, Wikidata is quickly emerging as a major knowledge base on the semantic web.

A user can get precisely the information *they* need using SPARQL and a connected triple-store. For example; A list of all airports with a name starting with 'X' located in China might not exist on the traditional web. Using the Semantic Web, however, a query retrieving this or almost any other information can be written. This allows the user to dictate what data they can get, instead of having to rely on some third-party to have published just what they need. However, as Dadzie et al. point out in their study, writing and executing structured language queries is difficult for the general public [4].

In order to write a query, a user must know a suitable triple-store[5], how to query it, and the SPARQL query language. Furthermore, information retrieval from triple-stores may be hard even *with* that knowledge. Since there is no schema, most triple-stores and many classes within a triple-store use different statements and identifiers for similar things. *The user would, therefore, have to be familiar with the data before making their query.*

[1]Which are also defined by URIs
[2]`https://musicbrainz.org/`
[3]`https://wiki.dbpedia.org/`
[4]`https://www.wikidata.org/wiki/Wikidata:Main_Page`
[5]A database that hosts Semantic Web data.

2.2 Information-Seeking Behaviour

A great deal of research has been made on *how* people acquire and search for information. Thomas D. Wilson presented in his 1997 work a general model for 'Information Behaviour' [5]. The model encompasses all human behaviour relating to information, but of most relevance to this work is *information-seeking behaviours*. Information seeking behaviour examines how users search, discover, and retrieve relevant data using an information system. This applies to all information systems, such as libraries, newspapers, or the internet. Wilson outlines four principal modes of information seeking; *Passive attention, passive search, active search, and ongoing search*. Wilson describes *Active Search* as modelling how individuals actively seek out new information to satisfy some *information need*.

Peter Pirolli & Stuart Card elaborate on how active search is performed in their 1999 work. They propose that human strategies for search and information retrieval can be modelled as a *foraging* behaviour [6]. *Optimal foraging theory* models how animals search for food, with an aim for maximizing food intake per unit of time. It describes how an animal will keep grazing on a patch of food until it is depleted, or it becomes more profitable to go to one with a higher yield. Between patches the animal relies on their senses, guiding it to a new patch of food.

Pirolli & Card propose information patches as the information science relatives of berry-patches [6]. In terms of information science, exploitation of a patch means to be informed by the data presented. An information patch may be *enriched* through activities like filtering, which increases the information density of the patch. Once all the information in a patch has been incorporated by the gatherer, they move on. If the information need has not yet been satisfied, the forager will follow an *information scent* in an attempt to find the next information patch.

Scent in terms of information science consists of 'imperfect information at intermediate locations', which the forager may use to determine which path to take in an information system. In a library, this may be the signs telling visitors what categories of books are in a given self. On the internet, this might be titles, descriptions, or icons hinting that relevant content might exist in that direction. This scent is what helps information seekers navigate, without which they would perform a random walk until a scent is picked up [6]. Information scent is, therefore, an essential instrument in guiding *foragers* to relevant sources of information.

Information scent is, however, of little use if the information source is *inaccessible*. Access is a fundamental requirement for information retrieval and may be blocked or inhibited in a number of ways [5]. The complexities of SPARQL are an inhibiting factors for access to the Semantic Web [4]. This threshold is hindering the mainstream from drawing advantage of the vast and detailed data the Semantic Web has to offer, creating a necessity for Semantic Web search engines [3, 4, 7].

2.3 Semantic Web Search Engines

Semantic Web Search Engines (SWSE) aim to simplify information retrieval on the Semantic Web. Users interact with these search engines through a graphical interface, and the search engines handle querying on their behalf.

Freitas et al. state that there is a trade-off being made between *usability* and *expressivity* in Semantic Web search engines [8]. Expressivity is a measure of how much control the user has over a query, or to what degree the user can specify what they want to retrieve. Usability is a measure of how user-friendly an interface is. As expressivity increases, usability tends to decrease. To illustrate this, Freitas et al. created a diagram of the expressivity-usability spectrum which can be seen in figure 2.1.

Pure SPARQL exists at the extreme expressive end of the spectrum, as it gives the user full control over their query, but is hard to use [4]. Natural language search as used by the Google search engine exists at the other extreme as the most usable, but least expressive interface.

Many Semantic Web search engines fall somewhere in the middle of this spectrum. This can in many cases be attributed to the functionality - or *category* - of the interface [8].

2.3.1 Categories of Semantic Web Search Engines

Semantic Web Search Engines are in this paper broadly divided in two categories; Query Building Interfaces (QBI), and Natural Language Interfaces (NLI). The two categories are separated by how a query is formed. Users of QBIs form queries using a step-by-step approach, gradually creating the full query. Queries in NLIs are on the other hand made by writing a query in plain english[6].

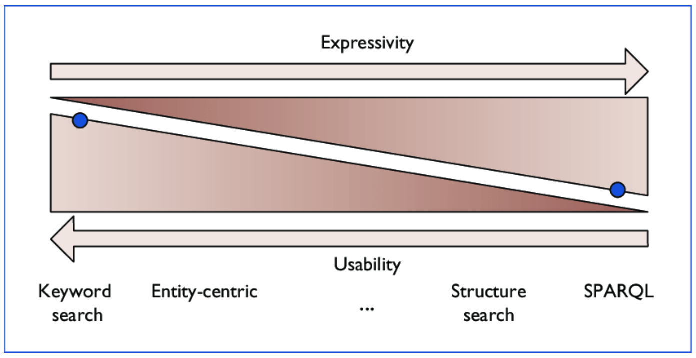


Figure 2.1: Freitas et al.'s diagram of the expressivity-usability spectrum, from their work [8]. Blue dots indicate the levels of usability and expressivity an ideal interface should have.

[6]Or another supported language

Natural Language Interfaces

Natural Language Interfaces (NLIs) allows the user to formulate queries with their everyday language. Mainstream search engines like Bing, Google, and Yahoo! all accept keyword- and full sentence searches through a simple text field. Many internet-users are familiar with this form of search interface, as searching is an integral part of the World Wide Web. The success and simplicity of these interfaces make them especially attractive to apply to the Semantic Web. NLIs are here grouped into two subcategories: Keyword- and natural language parsing search engines.

Keyword search engines accept or extract keywords from a query which is matched with related entities, in this context in a semantic-triple store [9, 10]. Match-scores are typically based on best *lexical* match, being the entity with the most similar string representation. Score is expressed as a continuous variable, determining how good a match it is. Some keyword-based search engines also attempt *disambiguation*, which determines whether a search for "House" is looking for the television series or a building. Disambiguating approaches may use a *discrete* matching system, determining whether an entity is *in* or *out* of the desired result set. Entities above a threshold score in continuous cases or 'in' in discrete cases, are returned to the user. Keyword-based approaches are, therefore, transparent and relatively simple.

Parsing NLIs attempt to formalize what a natural language input *means*, and use that semantic information to perform the search [11, 12]. This meaning can be extracted using several methods, such as grammatical rules or neural networks. The underlying semantics can then be translated into a query in a formal query language like SPARQL. This process of translation gives users the expressivity of a formal language, with the familiarity of their native tongue. How a query was interpreted and selection criterea set can however be complex, or in some neural cases impossible to tell. Neural approaches do however show great promise by being more flexible than traditional rule-based systems, returning more accurate result sets [13].

Query Building Interfaces

Query Building Interfaces can be loosely defined as interfaces guiding the user into creating a valid query. QBIs are divided into three subcategories. *Graphical/ Visual* query builders that employ graphic elements to build queries or guide the users [14–16]. *Form-based* approaches where users fill out a form which can be static or dynamically generated [17, 18]. Or, *faceted search* approaches, commonly used in online stores to navigate between facets or categories of products [19, 20].

Graphical interfaces rely on some manner of visualization to simplify the querying process. Interfaces like Affective Graphs [21] and Semantic Crystal [1] provide a graph-based visual query language, detailed in 2.4.1 and 2.4.7 respectively. These interactive graphs give a visual representation of the triple-patterns in the data sets. Searching is often done by constructing graphs with the same structure as that which the user is seeking. Since the query-graph is a direct visual representation of RDF, conversion to SPARQL is straight-forward.

Faceted interfaces present linked data by grouping data into categories based on the data *hierarchy*. In K-Search [22], the user chooses categories iteratively, gradually specifying the query. K-Search is detailed in 2.4. This is an approach widely employed in other structured information retrieval systems, such as e-commerce platforms [17]. By allowing the user to build their query by pointing and clicking, the user maintains an overview over where in the process they are, and what choices are available. Assuming the user has some knowledge of where the desired information can be found, navigating to it is as easy as following directions.

Form-based interfaces help users query linked data by providing a form to fill out. The fields of the form may be labelled and provide some context as to what is supposed to be filled in. Approaches using these interfaces often have dynamic forms that allow the user to start with a simple query, and gradually expand into a more complex one by including more relations and restrictions. Noteworthy examples are PepeSearch [23] and Corese [18]. PepeSearch is detailed in 2.4.5. Form-based interfaces provide a high level of guidance to the user, as the information required to pose a query is immediately visible.

2.4 Survey of Semantic Web Search Engines

This section presents a brief survey of notable Semantic Web search engines belonging to different categories. They are especially relevant to this study, as they have all been evaluated using the system usability scale, detailed in 2.5.1. Contributions, research and development methods, and system evaluation has been summarized to better relate these systems to the one presented in this report.

Esther Kaufmann and Abraham Bernstein were involved in creating the Ginseng (2.4.2), NLP-Reduce (2.4.4), Querix (2.4.6), and Semantic Crystal (2.4.7) interfaces [1]. The reason for including so many interfaces from Kaufmann & Bernstein was for their varied interface paradigms and use of the System Usability Scale.

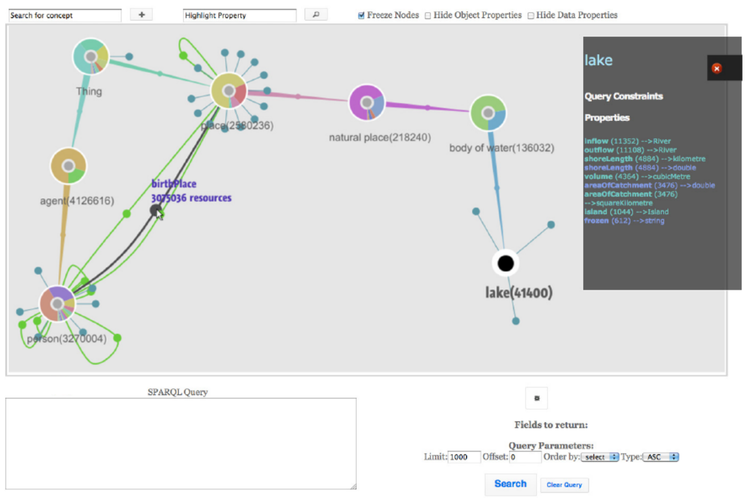
2.4.1 Affective Graphs

Affective Graphs is a query-building interface utilizing graph visualizations [21]. The tools' unique selling points is the point-and-click graph building interface and its consistent and well thought out aesthetic. A screenshot of the tool in use is shown in figure 2.2.

Contributions The main contributions made by Mazumdar et al. [21] are two-fold; First, a list of design principles for general linked data visualization, and Node-Link (graph-based) representations. Second, the Affective Graphs interface following the principles above.

Research and Development Methods A literature review was conducted from which Mazumdar et al. synthesized a list of design principles. These design principles were used as a guideline for the development of the Affective Graphs tool. Affective Graphs was developed in an iterative fashion where user feedback led to several re-designs.

Evaluation The final version of Affective graphs was evaluated in comparison to ten other interfaces and research prototypes. The interfaces were scored on how well they followed the design principles above, in addition to traditional metrics like the SUS [24] and user interviews. There were twenty participating users, comprised of ten laypeople and ten semantic web experts.



inflow (11352) -->River
outflow (11108) -->River
shoreLength (4884) -->kilometre
shoreLength (4884) -->double
volume (4364) -->cubicMetre
areaOfCatchment (3476) -->double
areaOfCatchment (3476) -->squareKilometre
island (1044) -->Island
frozen (612) -->string

Figure 2.2: The Affective Graphs interface in use.

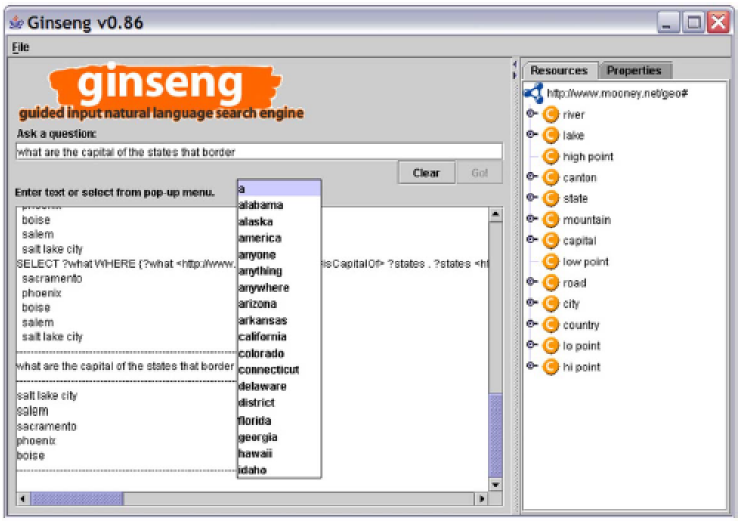
2.4.2 Ginseng

Ginseng, or ‘Guided Input Natural language Search Engine’, is a parsing-based natural language parser [25]. Ginsengs main feature is its restrictive, but guided, natural language search bar.

Contributions Ginseng has no built-in grammar, instead creating one from the currently loaded triple-store. This dynamic vocabulary makes Ginseng highly portable. To help users formulate queries in the restricted natural language, a predictive autocomplete dropdown menu appears while typing. The dropdown menu is visible in the screenshot of the tool in figure 2.3.

Research and Development Methods No research or development methodology was outlined by Bernstein et al. in [25].

Evaluation Ginseng was evaluated by automatically running 880 natural language queries from a geographical knowledge base. The performance measures chosen in this test was precision and recall. Numbers on how many queries ran out-of-the-box, how many worked without alterations, and how many that could not be formulated at all were also recorded. In the later publication by Kaufmann et al. [1], the system was evaluated using the SUS [24], and other usability related metrics such as time spent on query formulation.



```
what are the capital of the states that border
```

```
 boise
 salem
 salt lake city
SELECT ?what WHERE {?what <http://www.        isCapitalOf> ?states . ?states <ht
 sacramento
 phoenix
 boise
 salem
 salt lake city
-----------------------------------------------
what are the capital of the states that border
-----------------------------------------------
salt lake city
salem
sacramento
phoenix
boise
-----------------------------------------------
```

Figure 2.3: The Ginseng interface in use.

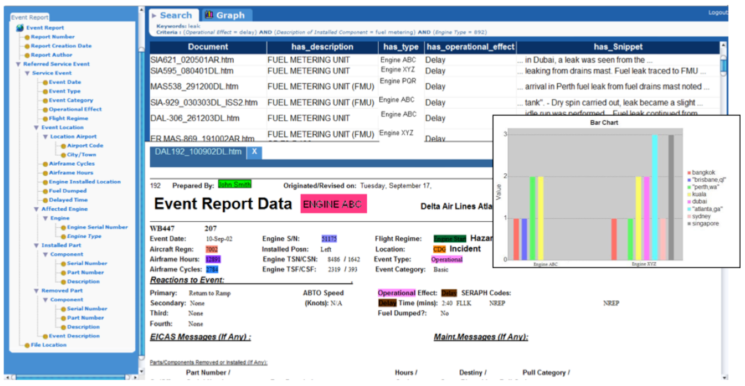
2.4.3 K-Search

K-Search is a form-based approach [22]. Its most unique feature is the combination of a free-text keyword search with structured semantic search. A screenshot of the interface is presented in figure 2.4.

Contributions The main contribution of Bhagdev et al. [22] is the development of their hybrid search technique. K-Search combines traditional information retrieval and semantic search approaches, retrieving results both based on document content and structured data. This gives more consistent performance in ontologies with highly heterogeneous data.

Research and Development Methods No research or development methodology was outlined by Bhagdev et al. in [22].

Evaluation K-Search was evaluated on both performance and usability measures. Precision and recall were used as a performance measure, where 21 predefined queries were tested on a proprietary triple-store. The usability tests, comprised of a questionnaire and an interview, commented on efficiency, effectiveness, and user satisfaction. The questions of the questionnaire were not disclosed. The usability tests were conducted using 32 'professional users' with relevant domain knowledge [22].



Keywords: leak
Criteria : (*Operational Effect* = delay) AND (*Description of Installed Component* = fuel metering) AND (*Engine Type* = 892)

Document	has_description	has_type	has_operational_effect	has_Snippet
SIA621_020501AR.htm	FUEL METERING UNIT	Engine ABC	Delay	... in Dubai, a leak was seen from the ...
SIA595_080401DL.htm	FUEL METERING UNIT	Engine XYZ	Delay	... leaking from drains mast. Fuel leak traced to FMU ...
MAS538_291200DL.htm	FUEL METERING UNIT (FMU)	Engine PQR	Delay	... arrival in Perth fuel leak from fuel drains mast noted ...
SIA-929_030303DL_ISS2.htm	FUEL METERING UNIT (FMU)	Engine ABC	Delay	... tank". - Dry spin carried out, leak became a slight ...
DAL-306_261203DL.htm	FUEL METERING UNIT	Engine ABC	Delay	idle run was performed... Fuel leak continued from
FR MAS-869_191002AR.htm	FUEL METERING UNIT (FMU)	Engine XYZ	Delay	

DAL192_100902DL.htm X

192 Prepared By: John Smith Originated/Revised on: Tuesday, September 17,

Event Report Data ENGINE ABC Delta Air Lines Atla

WB447 207

Event Date:	10-Sep-02	Engine S/N:	51175	Flight Regime:	Engine Start Hazar
Aircraft Regn:	7002	Installed Posn:	Left	Location:	CDG Incident
Airframe Hours:	12891	Engine TSN/CSN:	8486 / 1642	Event Type:	Operational
Airframe Cycles:	2784	Engine TSF/CSF:	2319 / 393	Event Category:	Basic

Reactions to Event:

Primary:	Return to Ramp	ABTO Speed		Operational Effect:	Delay SERAPH Codes:
Secondary:	None	(Knots):	N/A	Delay Time (mins):	2:40 FLLK NREP NREP
Third:	None			Fuel Dumped?:	No
Fourth:	None				

EICAS Messages (If Any) : *Maint.Messages (If Any):*

Parts/Components Removed or Installed (If Any):

Part Number / Hours / Destiny / Pull Category /

Figure 2.4: The K-Search interface in use.

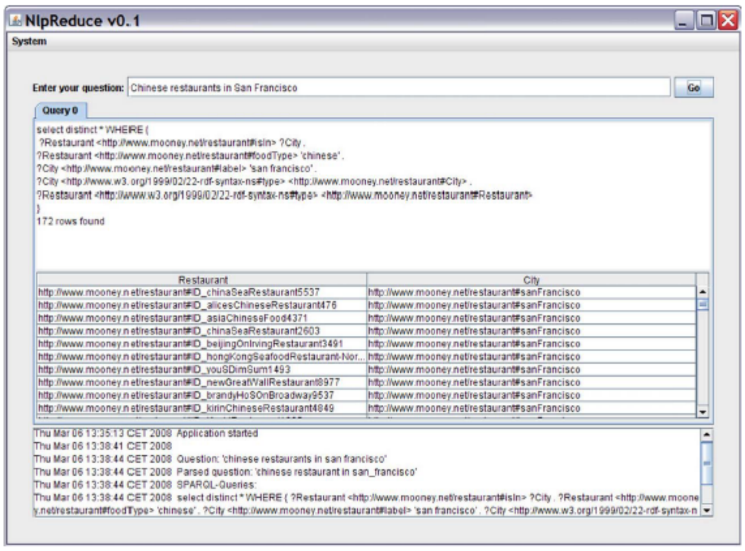
2.4.4 NLP-Reduce

NLP-Reduce is a parsing natural language interface [10]. It has a *reduced* set of NLP operators with a focus on portability and robustness to deficient input. The interface is presented in figure 2.5.

Contributions Kaufmann et al. present the NLP-Reduce interface as their main contribution. The tool is referred to as a 'reduced' NLP approach, as it only performs query expansion and stemming. Using the tokens returned through expansion and stemming, NLP-Reduce maps query terms to properties and entities in the underlying triple-store. This techinque is reportedly robust, maps queries well where triple-store labels are close to the natural language input.

Research and Development Methods No research or development methodology was outlined by Kaufmann et al. in [10].

Evaluation The tool was evaluated on performance and in a later study, usability as well. Precision and recall were recorded on two Mooney natural language ontologies. In the later publication by Kaufmann et al. [1], the system was evaluated using the SUS, and other usability related metrics such as time spent on query formulation.



Chinese restaurants in San Francisco

```
select distinct * WHERE {
 ?Restaurant <http://www.mooney.net/restaurant#isIn> ?City .
?Restaurant <http://www.mooney.net/restaurant#foodType> 'chinese' .
?City <http://www.mooney.net/restaurant#label> 'san francisco' .
?City <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.mooney.net/restaurant#City> .
?Restaurant <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.mooney.net/restaurant#Restaurant>
}
172 rows found
```

Restaurant	City
http://www.mooney.net/restaurant#ID_chinaSeaRestaurant5537	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_alicesChineseRestaurant476	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_asiaChineseFood4371	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_chinaSeaRestaurant2603	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_beijingOnIrvingRestaurant3491	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_hongKongSeafoodRestaurant-Nor...	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_youSDimSum1493	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_newGreatWallRestaurant8977	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_brandyHoSOnBroadway9537	http://www.mooney.net/restaurant#sanFrancisco
http://www.mooney.net/restaurant#ID_kirinChineseRestaurant4849	http://www.mooney.net/restaurant#sanFrancisco

```
Thu Mar 06 13:35:13 CET 2008  Application started
Thu Mar 06 13:38:41 CET 2008
Thu Mar 06 13:38:44 CET 2008  Question: 'chinese restaurants in san francisco'
Thu Mar 06 13:38:44 CET 2008  Parsed question: 'chinese restaurant in san_francisco'
Thu Mar 06 13:38:44 CET 2008  SPARQL-Queries:
Thu Mar 06 13:38:44 CET 2008  select distinct * WHERE { ?Restaurant <http://www.mooney.net/restaurant#isIn> ?City . ?Restaurant <http://www.moone
y.net/restaurant#foodType> 'chinese' . ?City <http://www.mooney.net/restaurant#label> 'san francisco' . ?City <http://www.w3.org/1999/02/22-rdf-syntax-n
```

Figure 2.5: The NLP-Reduce interface in use.

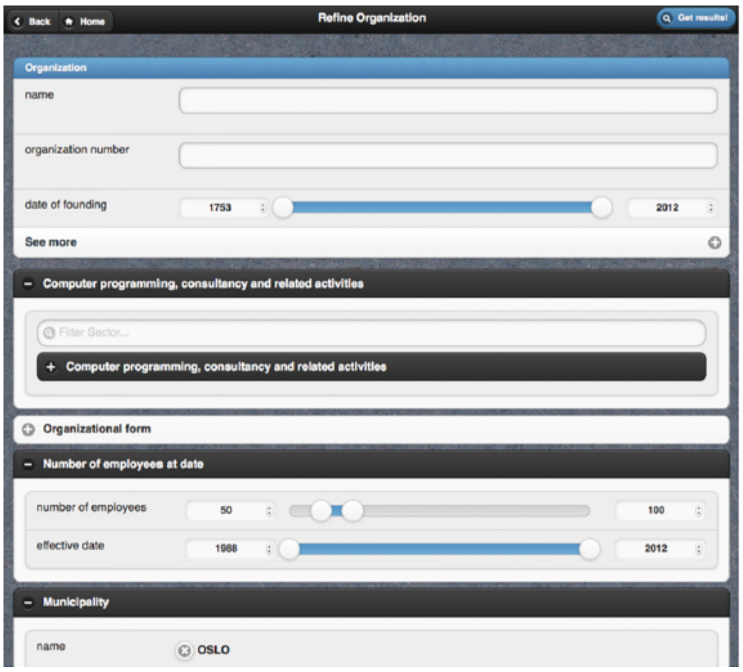
2.4.5 PepeSearch

PepeSearch is a form-based query builder [23]. The interface was developed with a focus on providing a usable and portable solution. A screenshot of the interface is presented in figure 2.6.

Contributions With PepeSearch, Vega-Gorgojo et al. seeks to enable mainstream users to query the semantic web. Their contribution is the PepeSearch tool, which maps RDF class properties to visual elements in a form. Thus, the users can at all times see the underlying data structure. This, in turn, makes the users more aware of what questions they can ask their data. The tool is also made to be completely portable, meaning that it can be used with any given SPARQL endpoint.

Research and Development Methods No research or development methodology was outlined by Vega-Gorgojo et al. in [23].

Evaluation The interface was measured on both performance and usability in the study. Performance was measured by the F-measure, the harmonic mean of precision and recall. Usability was tested with 15 participants using the System Usability Scale.



name
organization number
date of founding 1753 2012
Computer programming, consultancy and related activities
+ Computer programming, consultancy and related activities
Number of employees at date
number of employees 50 100
effective date 1988 2012
Municipality
name OSLO

Figure 2.6: The PepeSearch interface in use.

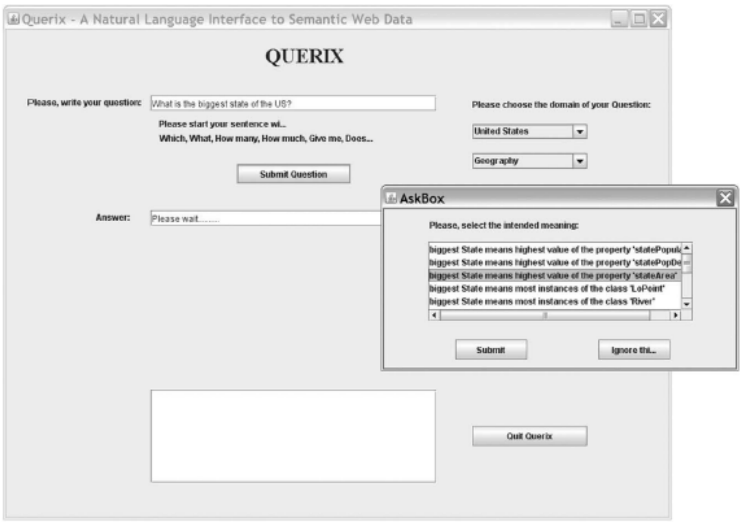
2.4.6 Querix

Querix is a parsing natural language interface [11]. It solves ambiguous queries through question answering. A screenshot of the tool in use can be found in figure 2.7.

Contributions The main contribution by Kaufmann et al. with their tool Querix is the use of question answering to solve ambiguities. Querix features an unrestricted natural language input field. However, natural language is hard to parse and often ambiguous. To solve the classic NLP issues, Querix uses a system of question-answering dialogue boxes. After a query has been posed, Querix will ask the user to specify what they meant by their query. This resolves ambiguities and helps parse the intent of a given query.

Research and Development Methods No research or development methodology was outlined by Kaufmann et al. in [11].

Evaluation The tool was evaluated on performance and in a later study, usability as well. Precision and recall were measured using the 2001 Mooney natural language geographical knowledge base. In the later publication by Kaufmann et al. [1], the system was evaluated using the SUS, and other usability related metrics such as time spent on query formulation.


Querix - A Natural Language Interface to Semantic Web Data

QUERIX

Please, write your question: What is the biggest state of the US?

Please start your sentence wi...
Which, What, How many, How much, Give me, Does...

Please choose the domain of your Question:
United States
Geography

Answer: Please wait.......

AskBox

Please, select the intended meaning:

biggest State means highest value of the property 'statePopul
biggest State means highest value of the property 'statePopDe
biggest State means highest value of the property 'stateArea'
biggest State means most instances of the class 'LoPoint'
biggest State means most instances of the class 'River'

Figure 2.7: The Querix interface in use.

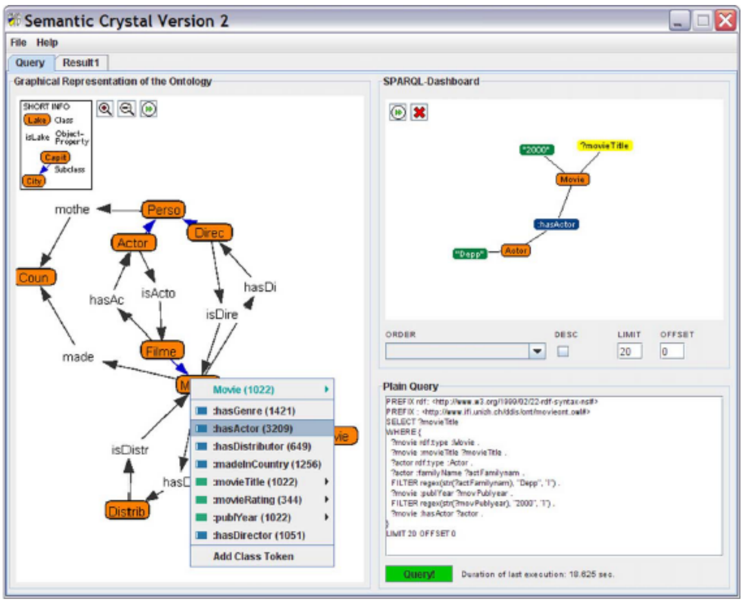
2.4.7 Semantic Crystal

Semantic Crystal is a graph-based query builder [1]. A graphical point-and-click interface was created to abstract the formal SPARQL. This interface is shown in the screenshot in figure 2.8

Contributions With the Semantic Crystal interface, Kaufmann et al. set out to support most of the expressivity of SPARQL while making it more usable for non-expert users. The main contribution is, therefore, their graphical query language, which allows users to point-and-click their way to a complete SPARQL query.

Research and Development Methods No research or development methodology was outlined by Kaufmann et al. in [1].

Evaluation The tool was evaluated on its usability. Kaufmann et al. [1] evaluate the system using the SUS, along with metrics like time needed to formulate a query, and comments on the design by test participants.



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.ifi.unizh.ch/ddis/ont/movieont.owl#>
SELECT ?movieTitle
WHERE {
  ?movie rdf:type :Movie .
  ?movie :movieTitle ?movieTitle .
  ?actor rdf:type :Actor .
  ?actor :familyName ?actFamilynam .
  FILTER regex(str(?actFamilynam), "Depp", "i") .
  ?movie :publYear ?movPublyear .
  FILTER regex(str(?movPublyear), "2000", "i") .
  ?movie :hasActor ?actor .
}
LIMIT 20 OFFSET 0
```

Duration of last execution: 18.625 sec.

Figure 2.8: The Semantic Crystal interface in use.

2.5 Usability

2.5.1 The System Usability Scale

The System Usability Scale - often referred to as the SUS - is a numerical measure of how user-friendly a system is. John Brooke (1996) and others developed it to provide a simple, standardized form for scoring a systems usability [24]. Though its' author refers to it as "'Quick and Dirty"', Bangor et al. report in their study that the scale captures the perceived usability score of a system very well [26].

The scale is a ten-question form with five responses for each question, ranging from 'Strongly Disagree' to 'Strongly Agree'[7]. The form, which can be seen in table 2.1, questions the user on the apparent simplicity or complexity of the system, whether it was enjoyable, and how easy or hard it was to use.

		Strongly disagree				**Strongly agree**
		1	**2**	**3**	**4**	**5**
1.	I think that I would like to use this system frequently.	○	○	○	○	○
2.	I found the system unnecessarily complex.	○	○	○	○	○
3.	I thought the system was easy to use.	○	○	○	○	○
4.	I think that I would need the support of a technical person to be able to use this system.	○	○	○	○	○
5.	I found the various functions in this system were well integrated.	○	○	○	○	○
6.	I thought there was too much inconsistency in this system.	○	○	○	○	○
7.	I would imagine that most people would learn to use this system very quickly.	○	○	○	○	○
8.	I found the system very awkward/cumbersome to use.	○	○	○	○	○
9.	I felt very confident using the system.	○	○	○	○	○
10.	I needed to learn a lot of things before I could get going with this system.	○	○	○	○	○

Table 2.1: The System Usability Scale by John Brooke et al. as used in this study.

[7]Also called a *Likert* scale.

Odd-numbered questions are asked in a positive tone, and even-numbered questions are posed in a negative tone. To account for this, even and odd questions are scored differently. The SUS score calculation as defined in [24] is as follows;

> *SUS yields a single number representing a composite measure of the overall usability of thesystem being studied. Note that scores for individual items are not meaningful on their own.*
>
> *To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4. For items 1,3,5,7,and 9 the score contribution is the scale position minus 1. For items 2,4,6,8 and 10, the contribution is 5 minus the scale position Multiply the sum of the scores by 2.5 to obtain the overall value of SU.*

The 'raw' SUS score is meant as a relative measure of usability. Sauro et al. in their quantitative study on usability scores, defines a grading curve for SUS scores which can be used as an absolute score [27]. This grading curve can be seen in table 2.2.

SUS Score Range	Grade	Percentile Range
84.1 - 100	A+	96-100
80.8 - 84	A	90-95
78.9 - 80.7	A-	85-89
77.2 - 77.1	B+	80-84
74.1 - 77.1	B	70-79
72.6 - 74	B-	65-69
71.1 - 72.5	C+	60-64
65 - 71	C	41-59
62.7 - 64.9	C-	35-40
51.7 - 62.6	D	15-34
0 - 51.7	F	0-14

Table 2.2: The SUS score grading scale as proposed by Sauro et al..

Bangor et al. found the mean SUS score for Web user interfaces to be 68.05 with a sample size of 1180 tests [26]. This allows the scale to be used to assess whether a system has greater or lesser usability than an average system, which is commonly used as a benchmark. This metric is therefore of particular relevance in this project, as the aim is to create a more usable Semantic Web interface.

This scale was used to score the systems evaluated in 2.4, making a comparison between them and Lister straight-forward. Furthermore, the mean SUS score for Web interfaces is well suited as a benchmark score for evaluating *hypothesis 5*.

2.5.2 Usability of Semantic Web Search Engines

According to the report by Berners-Lee et al. [3] and the survey by Dadzie et al. [4], the interfaces currently available are not usable enough for the casual end-user. Hachey et al. further report that only a few interfaces are ever tested using usability tests [7].

The general lack of usability test results makes it hard to gauge progress in the field and where future efforts should be placed. However, the seven interfaces reviewed in this paper have all been tested with the SUS. The test results gathered in table 2.3 substantiate the claims made by Berners-Lee and Dadzie, as only two out of seven interfaces tested above the SUS average.

All tests are done with users that are deemed to be *non-experts*. Of note is the fact that all tests except *PepeSearch* and *Querix* scored below the average usability score of 68.05. All the interfaces examined in the qualitative study in section 2.4 reported their own usability test scores, which can be seen in table 2.3. Ginseng (2.4.2), NLP-Reduce (2.4.4), and Semantic Crystal (2.4.7) were also tested by Elbedweihy et al.[28].

Tool	**System Usability Score**	**Tool type**
Affective Graphs	55.00*	QBI (Graph)
Ginseng	53.7*, 55.10†	NLI
K-Search	41.25*	QBI (Form)
NLP-Reduce	43.75*, 56.72†	NLI
PepeSearch	75.30°	QBI (Form)
Querix	75.73†	NLI
Semantic Crystal	61.25*, 36.09†	QBI (Graph)

Table 2.3: Usability tests of different SWSEs. Results marked with † are from [1], * are from [28], $^\circ$ is from [23].

Usability of Query Building Interfaces

QBIs split the query-process into several steps, guiding the user on the path to a complete query. A graphical representation of the querying process of a QBI can be seen in figure 2.9. While these steps guide the user, they can also be a burden on the usability of the interface. This was the case for several of the interfaces reviewed by Kaufmann et al. in their report [1].

The time required to pose a query generally goes up as the number of steps required to build a query increases[1]. Should the query then fail, or retrieve wrong results, the process has to start all over again. Five of ten questions in the SUS are about the perceived simplicity of an interface. It therefore stands to reason that a cumbersome querying and query-reformulation process could lead to poor usability scores.

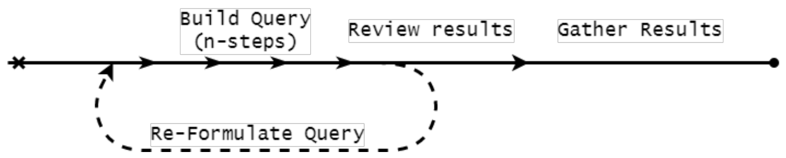


Figure 2.9: The Querying Process in a QBI

The layouts of QBIs are often closely tied to the data they provide access to. Faceted search engines show classes as a hierarchy, with tabs or flowchart-like diagrams. Graphical interfaces map shapes and connections to classes and relations. Form-based approaches adapt to the underlying data by simply displaying it; Either as columns in a table, items in a drop-down menu, or data-fields. Thus, in large heterogeneous ontologies, the QBIs tend to become overly complicated or overpopulated by visual elements.

The close connection between the number of visual elements in the interfaces and the underlying data makes QBIs scale poorly. The close relation between the data and the interface may also result in lower usability due to *leaky abstractions*. This means that the underlying triple-structure in some way *shines through* to the high-level interface. Leaky abstractions and poor scalability in SWSEs have been documented through usability tests and qualitative studies [1, 28, 29].

Finally, queries generally take longer to formulate in QBIs and are reported to be more tedious [1, 28]. This might be due to the number of steps required to build a given query.

Usability of Natural Language Interfaces

The querying process of a Natural Language Interface is more straightforward than that of the QBIs. A query is typed in the users natural language and submitted. Though some systems use query-answering dialogues to disambiguate, the querying process is generally as presented in figure 2.10.

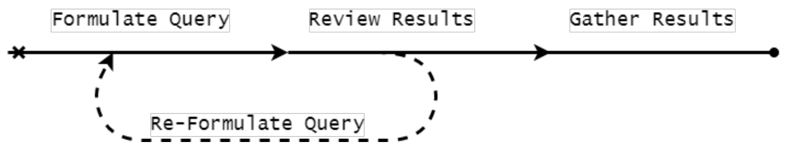


Figure 2.10: The Querying Process in an NLI

Though NLIs often are reported to be the most usable by casual users [1], the usability test results in table 2.3 do not seem to support this notion. The low usability scores in NLIs are likely not related to their interfaces, which typically are simpler than QBIs. It is the way NLIs work and their performance that leads to poor usability, the most notable example being the *habitability problem*.

The habitability problem occurs when users do not know whether the interface can understand their query, as they do not know what queries the system can parse [1, 8, 30]. Understanding whether a query was formulated wrong, or if the parser misunderstood it is a hard challenge. The goal of parsing NLIs is to be able to parse *any query*, but this has not yet been accomplished [31]. This may lead to confusion when users find that some queries are correctly parsed, while others fail silently.

2.6 Summary

The Background chapter has presented an overview of what the Semantic Web is, how people seek information, and an overview over and examples of search engines. It has presented results from reported usability tests of these interfaces and a short survey of SWSE's representing various user-interface paradigms. The key insights into the current state of- and issues in SWSE's are presented below.

The current state of Semantic Web Search Engines There are many different approaches to creating a usable interface to the Semantic Web. On one end of the expressivity-usability scale [fig 2.1] we find natural language interfaces, on the other end, pure SPARQL, and form-based approaches in the middle. The field of research is active, and it appears that knowledge of how to best approach Semantic Web search engines has not yet crystallized. Very few of the interfaces utilize usability measures like the System Usability Scale, or user driven development. Surveys of the field report that both usability and performance is currently too low for a general public audience, and that standard benchmarks are lacking [3, 4, 7].

Usability Issues in Semantic Web Search Engines Through a qualitative study of literature and projects regarding SWSEs, issues regarding usability were discovered. Poor interface usability was central in several studies. Specific obstacles mentioned were, for example; The habitability problem, interfaces being overly complicated, opaque in their functionality, or requiring some sort of background knowledge. These issues make it harder for a general audience to effectively use the interfaces, thereby limiting public access to the Semantic Web.

Chapter 3

Method

This chapter presents why a development project was undertaken, how the usability problem in semantic web search engines was mapped, the approach taken to solve the problem, and the methods used to evaluate the solution.

3.1 Research Methodology

To examine the research goals and test the hypotheses put forth in 1.2, a development project was undertaken. All aspects of the hypotheses were incorporated into the resulting system, which was tested in the testing phase. As the proposed novel system was explicitly designed to test the hypotheses, validity was ensured.

The system can be configured to work with any SPARQL endpoint. However, to provide a basis for testing, Wikidata was chosen. Wikidata has an active developer and data editor community, and a large amount of data spanning a multitude of domains. The generally high-quality and diverse domain data made the triple-store well suited for supporting a wide range of user queries, which was desirable for user testing.

3.2 Mapping The Problem

Analyzing a corpus of over 350 million SPARQL queries, Angela Bonifati, Wim Martens, and Thomas Timm (2019) found that almost 92% of all queries are SELECT queries [32]. The decision was therefore made to support SELECT queries exclusively, as this greatly reduces the complexity of the interface.

A further focus was made on promoting data exploration through *list-retrieval* rather than focusing on single entity retrieval. List retrieval is defined here as the process of retrieving all *member instances* of a *class*. The search is therefore entity-based, as a search for the class-entity 'Mountain' retrieves entities like 'Mount Everest' and 'Galdhøpiggen'. Semantic data-stores are well suited for this category of information retrieval, as hierarchies and class membership are explicitly defined as part of the triple-structure of the data.

To maximize both the *usability* and *expressivity* of the solution, a *hybrid* system was proposed; Natural language keyword search was implemented for discovering classes, and form-based approaches are used for exploring and interacting with the retrieved class-instance dataset. This combines the usability and familiarity of natural language search, with the high expressivity of form-based query builders.

The interface was developed with the *information foraging* model as a guiding principle. A user confronted with an empty result set will lose their *information scent*. With no data, it is hard to know whether the query or the system itself was at fault, and no clues are given on which path to take next. An interface displaying *too much* information might, on the other hand, overwhelm the user, losing any information scent in the crowd. These issues were addressed by exploiting the topology of linked data, ensuring that *no blank results* can appear and that a *high level of accuracy* is maintained in the class search results.

Information patches were also considered. Users should be able to interact with their data without query reformulation, thereby *enriching* their information patch. Less querying and more exploiting also increases the rate of information gain, a central goal of information foragers [6].

A diagram of how interaction with such a system might take place was made. The intention was to outline how a user would interact with the hybrid system while keeping the path from start to the desired result as short as possible. It was hypothesized that by minimizing the number of steps from the user started interaction to them having their desired results would improve usability, as observed in natural language interfaces 2.5.2. The resulting process-diagram (fig 3.1) served as a basis for further development.

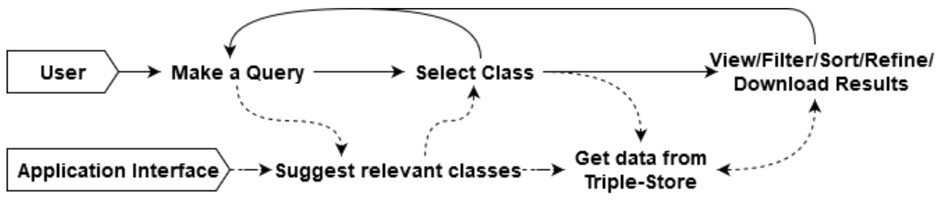
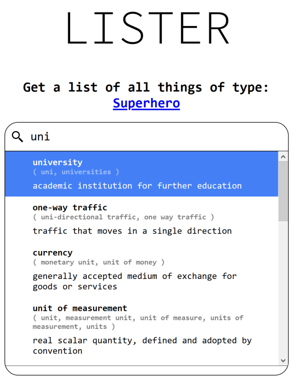


Figure 3.1: A process map showing the operation of the proposed list-retrieval system, where solid lines represent user actions and dashed lines code exectuion.

3.3 Approach

With the research questions, hypotheses, and process map in mind, the Lister system, as shown in figure 3.2 was developed. This section details how the system sets out to answer the research questions and create a foundation for testing the hypotheses. More details on the search engine's architecture and functionality are presented in chapter 4.



LISTER

Get a list of all things of type:
Superhero

uni

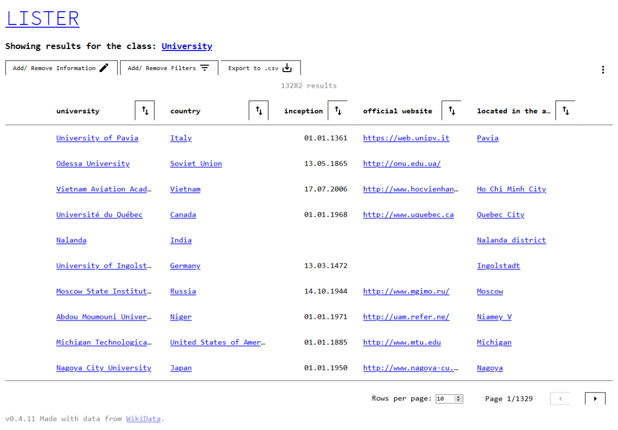
university
(uni, universities)
academic institution for further education

one-way traffic
(uni-directional traffic, one way traffic)
traffic that moves in a single direction

currency
(monetary unit, unit of money)
generally accepted medium of exchange for goods or services

unit of measurement
(unit, measurement unit, unit of measure, units of measurement, units)
real scalar quantity, defined and adopted by convention

(a) The Lister landing page, showing a query under formulation.



LISTER

Showing results for the class: University

13282 results

university	country	inception	official website	located in the a…
University of Pavia	Italy	01.01.1361	https://web.unipv.it	Pavia
Odessa University	Soviet Union	13.05.1865	http://onu.edu.ua/	
Vietnam Aviation Acad…	Vietnam	17.07.2006	http://www.hocvienhan…	Ho Chi Minh City
Université du Québec	Canada	01.01.1968	http://www.uquebec.ca	Quebec City
Nalanda	India			Nalanda district
University of Ingolst…	Germany	13.03.1472		Ingolstadt
Moscow State Institut…	Russia	14.10.1944	http://www.mgimo.ru/	Moscow
Abdou Moumouni Univer…	Niger	01.01.1971	http://uam.refer.ne/	Niamey V
Michigan Technologica…	United States of Amer…	01.01.1885	http://www.mtu.edu	Michigan
Nagoya City University	Japan	01.01.1950	http://www.nagoya-cu.…	Nagoya

Rows per page: 10 Page 1/1329

v0.4.11 Made with data from WikiData.

(b) The Lister result page, showing the entities of a class.

Figure 3.2: The two main views of the Lister search engine.

R.Q./H. 1: Interactive Techniques for Query Formulation By using a familiar search-bar concept for querying, users immediately know how to start their search. As the user types their query, predictive class suggestions appear below, guiding the user to the right query formulation while providing an *information scent*. This *Class-Search* process is crucial for guiding the user to the right class, and subsequently, the right data. The class search is implemented using a double index system (4.3) that maps queries to a corresponding class, either directly or by finding an entity in the query from which a class can be extracted.

R.Q./H. 2: Interactive Techniques for Data Exploration The data table is a ubiquitous representation format for data. It is also an ideal format for displaying, interacting with, and exploring semantic web data. Each row is made to represent one entity, with columns representing the properties of that entity. By using the table format, users who are entirely unfamiliar with the semantic web can still make sense of and work with the data. Entity URIs are formatted as links, using their human-readable titles as link texts. The user may add or remove columns, rank, filter, or download their data. These interactive elements enriches the *information patch* the data represents. They further enable users to play with and get an overview of the data presented in a user-friendly way, while still facilitating exploration through linked data.

R.Q./H. 3: Exploiting Data Topology for Result Ranking By pruning all classes without member entities, it is ensured that no query returns an empty result set and providing an *information scent* for the user. As the data-source is highly homogeneous, many unrelated classes and entities have a high textual similarity. When searching for classes, results are therefore ranked based on several factors; lexical similarity, number of member entities, properties, and inbound site links. This 'importance' score is similar to what search engines do[1] on the traditional web, keeping the results most likely to be relevant at the top of the result list.

R.Q./H. 4: Exploiting Data Topology for Explorative Search When a user is unfamiliar with a database or unsure of what their desired class is named, they might try searching for an entity of that class. By exploiting the class membership topology, the classes of entities related to their query are returned. When exploring a class, users might not be sure of what properties can be added to the retrieved entities. This is solved by pre-fetching all available properties of a subset of the class entities. When the results have loaded, the four most popular properties have already been added, providing an overview of the class. When adding more information, the user is presented with properties that are guaranteed to be relevant, ordered by popularity[2]. This makes data exploration effortless, avoiding guesswork as to what properties are relevant.

R.Q./H. 5: Iterative Development By following an iterative development process, usability issues are caught early, and user feedback can be incorporated into the system. This is crucial for discovering how users model the system, how they interact as opposed to how they *should* interact, and in verifying that fixes from the last iteration improved the system. How the iterative development was done in practice is further detailed in 3.4.1.

3.4 Evaluation Methods

Qualitative *and* quantitative methods were used to test the system in regards to the hypotheses put forth in 1.2. The qualitative data gathering took place in two phases during the development of the project 3.4.1, and as part of the quantitative testing at the end of the project 3.4.2. The qualitative methods were chosen to get more detailed user feedback through user interviews and on-site testing. No dataset for testing class-search was found, and traditional measures like precision and recall were not the focus of this project. Retrieval performance of the class-search functionality was therefore evaluated with the qualitative feedback from user-tests.

[1]Using PageRank was considered. However, the transfer of importance from classes of a high degree to those of low was considered undesirable in list-retrieval.

[2]Defined as the percentage of entities *of that class* having that property.

User testing, as part of an iterative development process, is a way to gather qualitative data about the system in development and how the user experience changes over time. Detailed feedback from how the user interacts and responds to the system can be gathered in the interview-like setting. This allows the developer to test hypotheses early, and to *steer* the project in the right direction.

The System Usability Scale was chosen as the project's *quantitative* part. The scale was chosen both for its widespread use in usability testing, and to be able to compare the usability of this project to the ones reviewed in 2.4.

3.4.1 User Testing

The user-tests in this report were modelled after those described by Jake Knapp, John Zeratsky, and Braden Kowitz in their 2016 book 'Sprint' [33]. A user-test is a setting where the user tries to solve several tasks using the system or product that is to be tested. The user is supervised by a facilitator that notes how the user reacts to- and interacts with the system. This test was designed to examine the validity of *hypotheses 1—4*.

Data was collected as the project progressed by performing two separate rounds of user testing. The first iteration was conducted in March using the prototype, where five users participated. Incorporating the feedback and creating the next version of the system took longer than expected due to the coronavirus pandemic forcing the author to relocate. The second iteration of user testing was conducted during the first half of May, with the subsequent development being concluded by the end of May.

Test Set-Up Performing a user test requires a test subject, a prototype to test, and a test facilitator. The user test should take place in a quiet and secluded environment, free of distractions. The test is set up so that the facilitator can see both how the test subject interacts with the prototype and how the subject reacts during the test. Notes can be taken during the test or later by recording the session.

The test is performed by the test subject solving tasks using the prototype. The facilitator should not interfere or provide feedback unless the user gets stuck. The facilitator should also take care to inform the user that it is the system that is being tested, not the user, and that honest feedback is essential. Participants were also informed that their test result would be anonymized, and no personal information saved.

To gather as many test subjects as possible, and in light of the public health authorities advice to minimize social contact at the time of writing, online testing was conducted. Remote testing was done using an online conferencing tool of the test subjects choosing, with the user completing the test while sharing their screen. The test consisted of the user testing the two main aspects of the prototype; The class-search and the data interaction. This was done through two tasks consisting of first finding the relevant class and subsequently exploring the data it contained.

3.4.2 System Usability Scale

The usability scale tests were performed unsupervised online from the 31st of March until the 5th of June. During that period, a total of 86 people completed the tests successfully. This test was designed to examine the validity of *hypotheses 1—4* qualitatively, and to quantitatively measure the SUS score to test *hypothesis 5*.

Test Set-Up As with the user tests described in 3.4.1, the usability tests were also held online. Several steps were taken to ensure the ethicality of the test; The data were anonymously collected through Google forms, requiring no personal information about the subject. It was explicitly stated that the test was made to evaluate the system and not the user. Finally, the test was voluntary, ensuring that all test participants consented to their participation in the study.

The tools reviewed in 2.4 were evaluated with casual users. A screening question was therefore formulated to ensure that Lister was tested on the same basis. Before starting the test, users needed to indicate their skill in using structured languages like SPARQL or SQL for accessing databases.

The usability test consisted of two parts; First, a task sheet with problems to solve using the project software Lister. Each task was on a separate sheet, and users were encouraged to go to the next page if they felt they had solved the task, or gotten stuck. After each task was completed, the user was shown detailed instructions on how the task could have been solved. This was to make sure that the user would able to finish the test and reliably rate the system.

After the tasks, the user was asked to fill in the System Usability Scale, as presented in 2.5.1. Further data gathering was done after the SUS test as not to bias the results. The users were subsequently asked whether they successfully finished the test and whether they found any bugs or had any comments or feedback on the system. This final part was voluntary and aimed at getting qualitative feedback on the performance and design of the final prototype of Lister.

Chapter 4

Implementation

This chapter details how users may operate the Lister search engine, how the code base is structured, functional details of the implementation, and what technologies were chosen.

4.1 Operation of the Lister Search Engine

4.1.1 Overview

The system consists of three different ‘views’; The home page, the class-search results page, and the list-search result page. These views aid the user in the two main functions of the Lister interface, selecting the right class (4.1.2), and exploring and exploiting their data (4.1.3). An overview of how they connect to each other can be seen in the sitemap diagram 4.1.

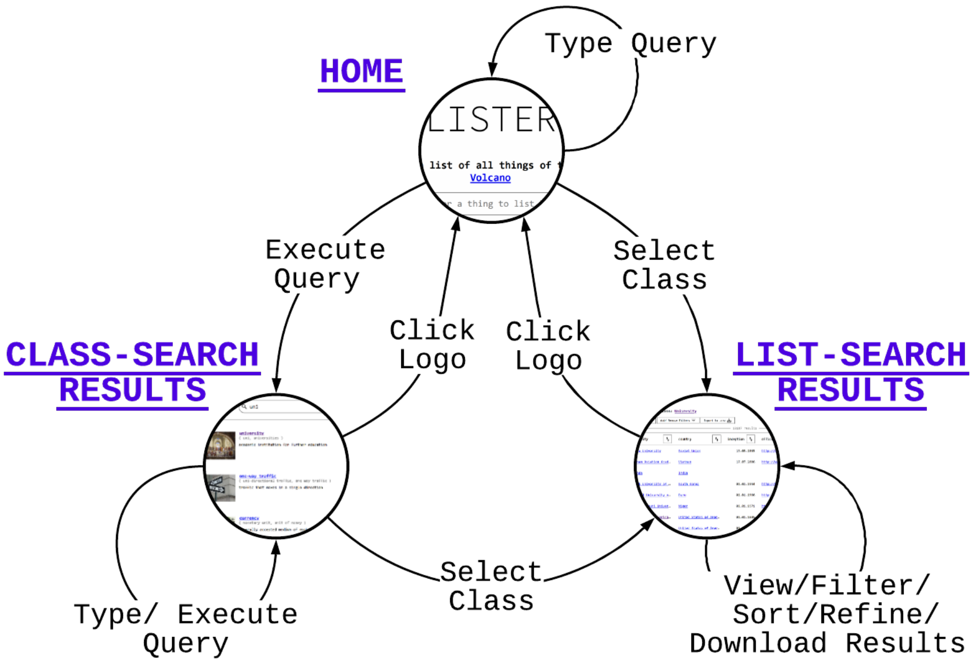


Figure 4.1: A Sitemap of the Lister Search Engine.

4.1.2 Selecting a Class

Upon navigating to the Lister home page[1] users are presented with a search bar and a carousel showing example queries. Users receive suggested classes in a result box below the search bar as they type their query. By selecting one of the suggestions or one of the example queries, they are taken to their results on the list-result page. By executing their query, the user is taken to the class-search result page.

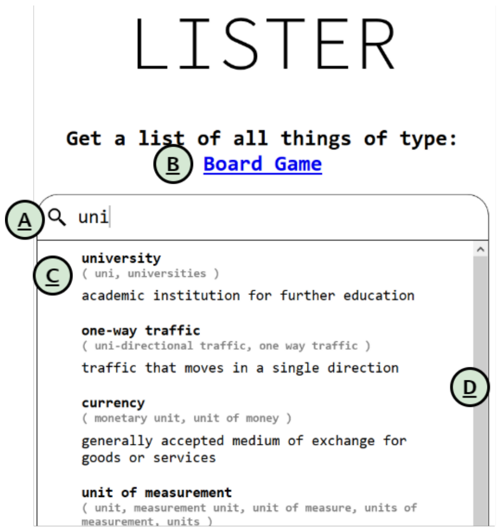
The class-search results page displays the class suggestions in a larger format. It includes picture representations of the different classes. Its functionality is otherwise the same as the landing page, and selecting a class here takes the user to the list-result page.

The elements of the landing and class-search result page and their functions are as follows, each item in 4.1 corresponding to a label in figure 4.2 and 4.3:

Label	Element	Function
A	Search Bar	Accepts user input and queries for corresponding classes.
B	Example Carousel	Rotates between example queries, which lead to their respective result sites when clicked.
C	Search Result	A result for the user query in (A), which leads to its' results page when clicked.
D	Result List	A list containing up to ten relevant result items.

Table 4.1: Description and functionality of landing and class-search results page elements labled in figures 4.2 and 4.3

[1]At the time of writing that is `https://folk.ntnu.no/matsjsk/`



Get a list of all things of type:
Board Game

uni

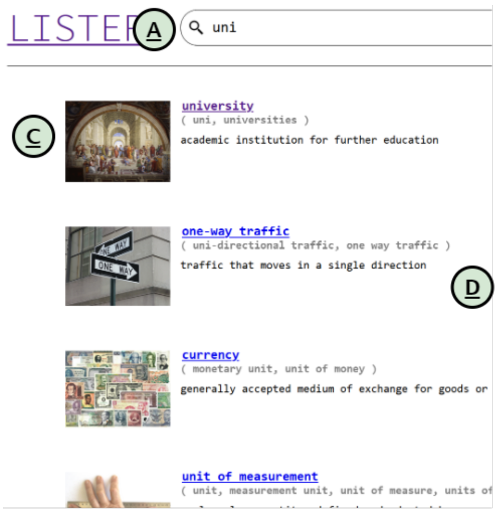
university
(uni, universities)
academic institution for further education

one-way traffic
(uni-directional traffic, one way traffic)
traffic that moves in a single direction

currency
(monetary unit, unit of money)
generally accepted medium of exchange for goods or services

unit of measurement
(unit, measurement unit, unit of measure, units of measurement, units)

Figure 4.2: The Lister landing page, showing a query under formulation.



uni

university
(uni, universities)
academic institution for further education

one-way traffic
(uni-directional traffic, one way traffic)
traffic that moves in a single direction

currency
(monetary unit, unit of money)
generally accepted medium of exchange for goods or

unit of measurement
(unit, measurement unit, unit of measure, units of

Figure 4.3: The Lister class search result page, showing suggested classes for a query.

4.1.3 Data Interaction

Once the user has selected their desired class, they are redirected to the list-search results page. This page displays the member entities of a class in a table format. It is automatically populated with the properties most used with entities of that class, as detailed in 4.4.2. The results are retrieved from a SPARQL endpoint connected to an online triple-store.

The user may interact with their data using a number of functions; Browsing pages, adding or removing information[2], filters, ranking- or downloading their data, or exploring linked URI's. They may also extend their search to include subclasses and visit URI's belonging to entities and classes.

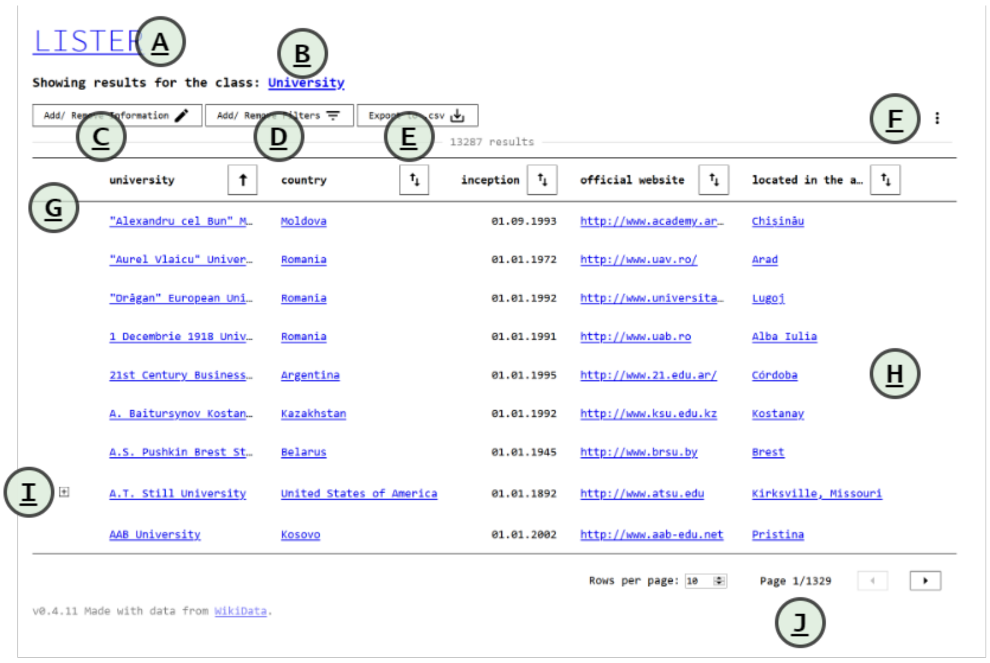
Results page

The elements of the results page and their functions are as follows, each item in 4.2 corresponding to a label in figure 4.4:

Label	Element	Function
A	Logotype	Links back to the landing page
B	Class URI	Links to the URI of the selected class
C	Add/ Remove Information	Button to open add/remove Information modal.
D	Filter Columns	Button to open filter columns modal.
E	Export to `.csv`	Button download data in a `.csv` format.
F	Search Options	Options to; Extend search to subclasses, expand/-group duplicate rows, and paginate data
G	Table Header	Contains column names and buttons for sorting, showing the most common columns by default.
H	Table Rows	A row of data representing an entity and its relations in a triple-store.
I	Expandable Row	Groups multi-valued columns to a single entity, detailed in 4.4.1.
J	Pagination	Page counter and last and next page buttons, controlling data offset.

Table 4.2: Description and functionality of result page elements labled in figure 4.4.

[2]This wording was chosen over 'Properties' by request in user-testing.



Showing results for the class: University

13287 results

university	country	inception	official website	located in the a…
"Alexandru cel Bun" M…	Moldova	01.09.1993	http://www.academy.ar…	Chisinău
"Aurel Vlaicu" Univer…	Romania	01.01.1972	http://www.uav.ro/	Arad
"Drăgan" European Uni…	Romania	01.01.1992	http://www.universita…	Lugoj
1 Decembrie 1918 Univ…	Romania	01.01.1991	http://www.uab.ro	Alba Iulia
21st Century Business…	Argentina	01.01.1995	http://www.21.edu.ar/	Córdoba
A. Baitursynov Kostan…	Kazakhstan	01.01.1992	http://www.ksu.edu.kz	Kostanay
A.S. Pushkin Brest St…	Belarus	01.01.1945	http://www.brsu.by	Brest
A.T. Still University	United States of America	01.01.1892	http://www.atsu.edu	Kirksville, Missouri
AAB University	Kosovo	01.01.2002	http://www.aab-edu.net	Pristina

Rows per page: 10 Page 1/1329

v0.4.11 Made with data from WikiData.

Figure 4.4: The Lister landing page before user interaction.

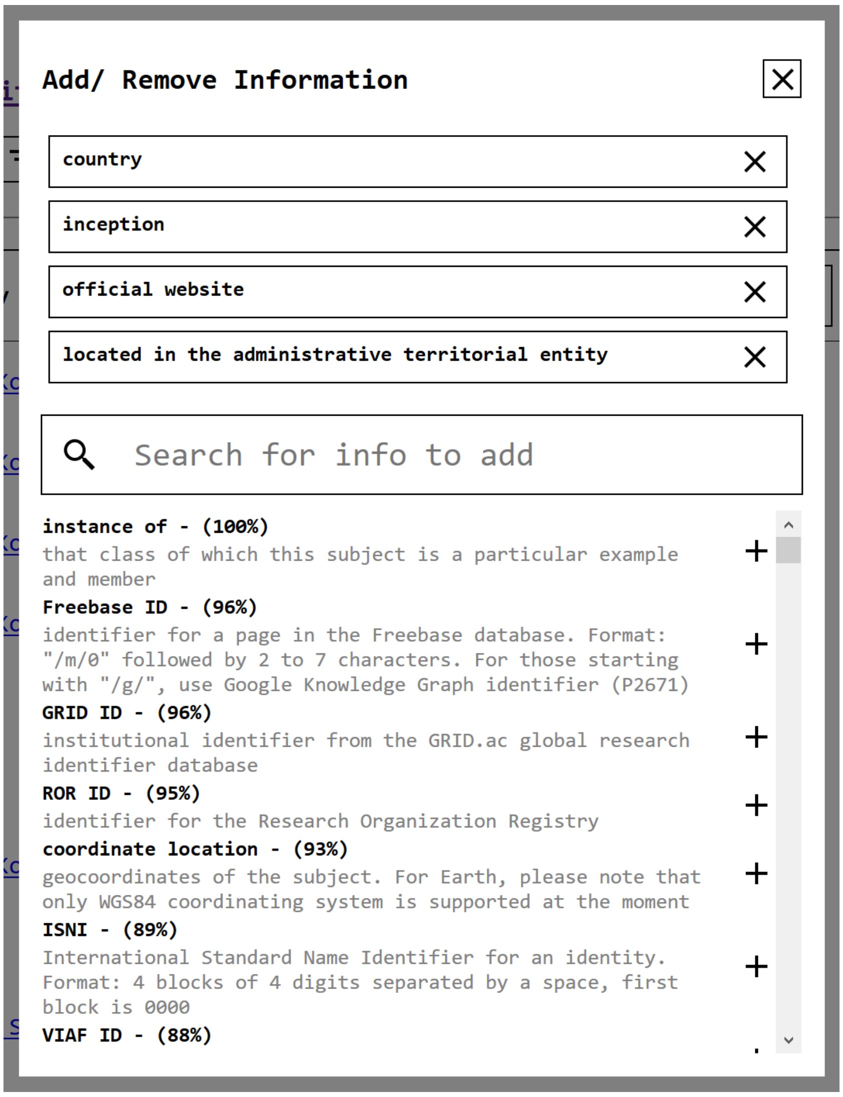
Adding and Removing Information

The user can add or remove information related to their listed entities, done by pressing the "Add/Remove Information" button. This function affects the properties applied to the members of the selected class. Information, hereafter referred to as properties, are removed by clicking on one of the *property chips* at the top of the modal (4.5 A), and added by clicking on a property item (4.5 D) in the column menu (4.5 E). The properties are ordered by decreasing frequency, where the frequency is the number of times that property appears with an instance of the selected class. The user may browse the property menu (4.5 E) or search for a property using the search bar (4.5 C). The modal may be closed by clicking the close button (4.5 B) or clicking outside the modal.

The elements of the add/ remove information and their functions are as follows, each item in 4.3 corresponding to a label in figure 4.5:

Label	Element	Function
A	Column Chips	Shows currently selected columns. Clicking a chip removes the column.
B	Close Button	Closes the modal.
C	Search Bar	Search bar for the column menu.
D	Column item	Shows a title, frequency, and description of a column. Clicking the column item selects the column.
E	Column menu	Lists the available column items by descending order of frequency.

Table 4.3: Description and functionality of result page elements labled in figure 4.5.



Add/ Remove Information

country

inception

official website

located in the administrative territorial entity

Search for info to add

instance of - (100%)
that class of which this subject is a particular example and member

Freebase ID - (96%)
identifier for a page in the Freebase database. Format: "/m/0" followed by 2 to 7 characters. For those starting with "/g/", use Google Knowledge Graph identifier (P2671)

GRID ID - (96%)
institutional identifier from the GRID.ac global research identifier database

ROR ID - (95%)
identifier for the Research Organization Registry

coordinate location - (93%)
geocoordinates of the subject. For Earth, please note that only WGS84 coordinating system is supported at the moment

ISNI - (89%)
International Standard Name Identifier for an identity. Format: 4 blocks of 4 digits separated by a space, first block is 0000

VIAF ID - (88%)

Figure 4.5: The modal for adding or removing columns.

Filtering Columns

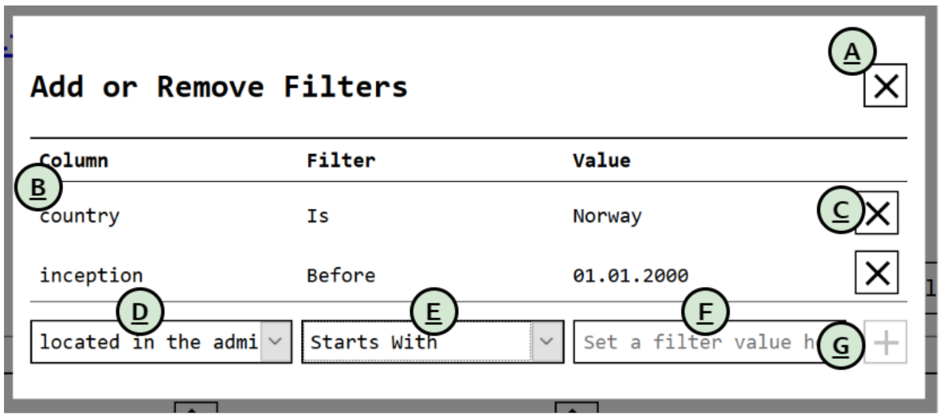
The result rows may be filtered by the values of their respective properties. The user does this by clicking the "Add/ Remove Filters" button and specifying a filter. Filters are specified by first selecting the property[3] that the filter should be applied to using the "Select Column" (4.5 D) select menu. Selecting a column populates the "Select Filter" select menu (4.5 E) with available filters appropriate to the data type of the selected column. If the user has chosen a filter that requires a value, the "Set filter value" field (4.5 F) will accept user input. The input element changes depending on the datatype, making it easier for the user to provide the appropriate input. After a filter value has been set, the user may add the filter by clicking the add filter button (4.5 G) or pushing the enter key.

Filters that have been added can be seen in the applied filters table (4.5 B). They may be removed by clicking the "X" button (4.5 C) corresponding to the row of the filter that the user wishes to remove.

Label	Element	Function
A	Close Button	Closes the modal.
B	Applied Filters Table	Lists currently applied filters.
C	Remove Filter Button	Removes the filter on the same row.
D	Column Select	Opens a select menu populated with the currently selected columns.
E	Filter Select	Opens a select menu populated with filters corresponding to the datatype of the selected column.
F	Value Input	Accepts user input specifying the value of a filter.
G	Add Filter Button	Applies the configured filter.

Table 4.4: Description and functionality of result page elements labled in figure 4.5.

[3]Referred to here as ‘Column’



Add or Remove Filters

Column	Filter	Value
country	Is	Norway
inception	Before	01.01.2000
located in the admi	Starts With	Set a filter value h

Figure 4.6: The modal for adding or removing filters.

4.2 System Architecture

4.2.1 Overview

The Lister system consists of a client-side web application that communicates with a server-side index and a SPARQL endpoint, as shown in figure 4.7. The front-end handles user interaction and data requests to the back end and the configured SPARQL endpoint. The Back-end has several endpoints facilitating full-text search for classes and entities, in addition to convenience functions such as entity-ID lookup and class of entity retrieval.

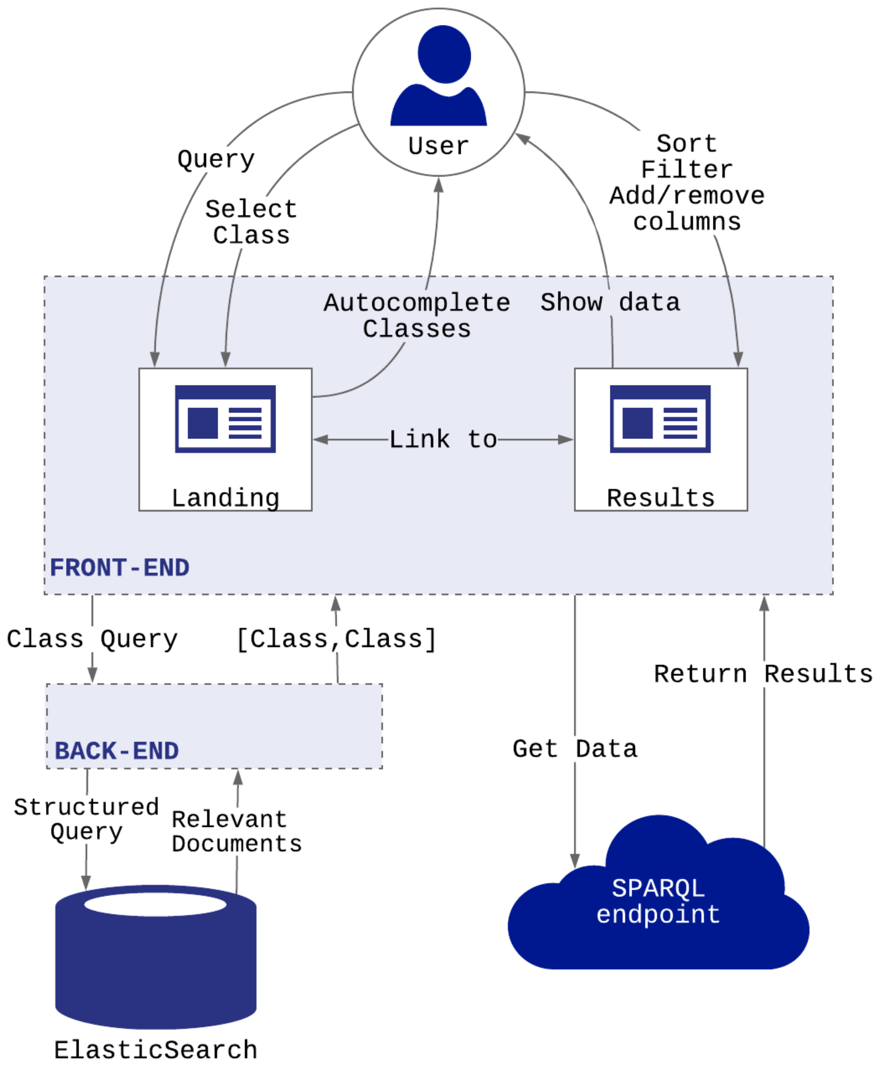


Figure 4.7: A high-level architecture of the Lister system, showing user- and internal interaction.

4.2.2 Front-End

The front end consists of a landing page[4], a result page, a request handler, and various utility functions. As the user interacts with the landing- and result-pages, the state of the web app changes and new data is fetched from the back-end or an external SPARQL endpoint. An overview of how the different pages, classes, and functions fit together can be seen in figure 4.8.

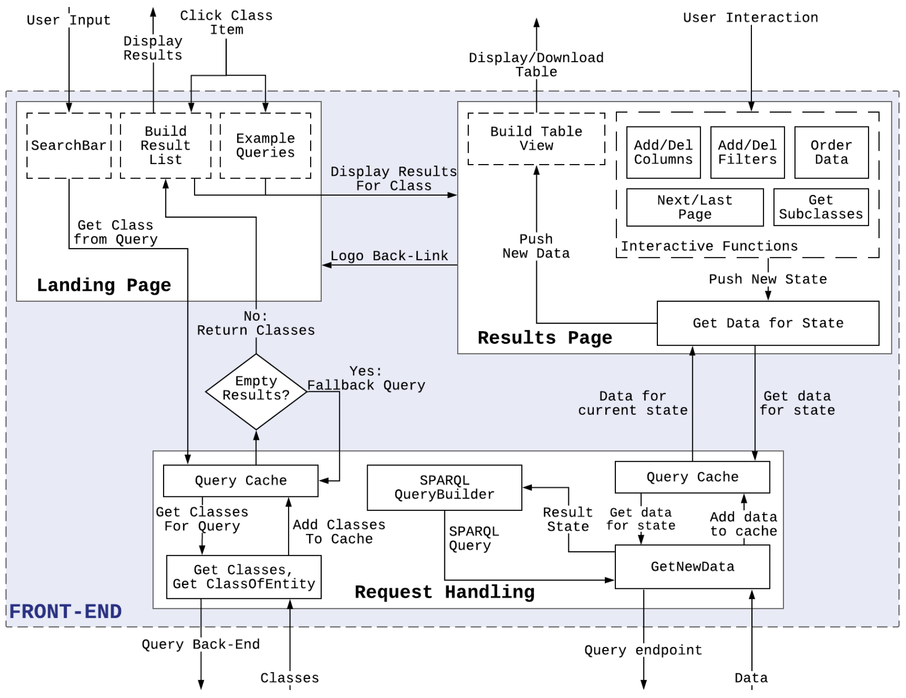


Figure 4.8: An overview of the different internal and external interactions of the front-end. Functions and classes are marked with solid outlines.

Landing Page

The landing page consists of a header, a small tagline with a roulette of example queries, and a search bar and its' corresponding results. It communicates only with the request handler, which fetches classes in response to a user query. Results from queries and the example queries link to their respective results pages.

[4]The class-search results page is functionally similar to the landing page, and is therefore omitted from the following figures and texts.

Results Page

The results page consists of a back-linking logo, a data-table, and several functions to facilitate interaction with the data-table. The results page fetches new data as the user changes the state of the table. State changes by adding or removing columns, filters, or navigating to the next page of results. New data coming in triggers a re-render of the table view if necessary.

Request Handling

The request handler collects the various functions needed for getting, cleaning, and responding to the network requests required for class-search and entity retrieval. All network requests are cached for the duration of the user session, saving network and computing resources, and allowing the user to view previous results quickly.

The class-search gets its' data from the ElasticSearch cluster[5]. Any request is first sent to the class-index, looking for a matching class. Should none be found, a fallback query is made to the `ClassesOfEntity`. This function retrieves and returns the classes of the top ten entities related to a query. This ensures that a wide range of queries gets good results.

In order to retrieve data for the result page, a SPARQL-query must be built and sent to an endpoint. The query is built based on the internal state of the data table. This SPARQL query follows standard syntax but uses WikiData specific property types. To query an endpoint other than WikiData, some reconfiguring would be necessary.

4.2.3 Back-End

The back-end consists of a Node.JS API and two ElasticSearch Indices, a diagram of which is presented in figure 4.9. The Node API facilitates communiation between the front-end and ElasticSearch cluster.

The primary purpose for the Node API is to protect the ElasticSearch cluster from malicious code injection and DOS attacks. However, it also provides convenient functions like cluster- and back-end health checks[6] and a service to resolve entity-search into relevant classes.

The ElasticSearch cluster contains two indices; One for all class-entities, and one for all entities. Management of the cluster is done either with the console on the server or through the Kibana dashboard.

[5]This service was first implemented in pure SPARQL to keep the Lister Web App as lightweight and portable as possible, but proved impractical due to long processing times.

[6]Not indicated on the back-end diagram

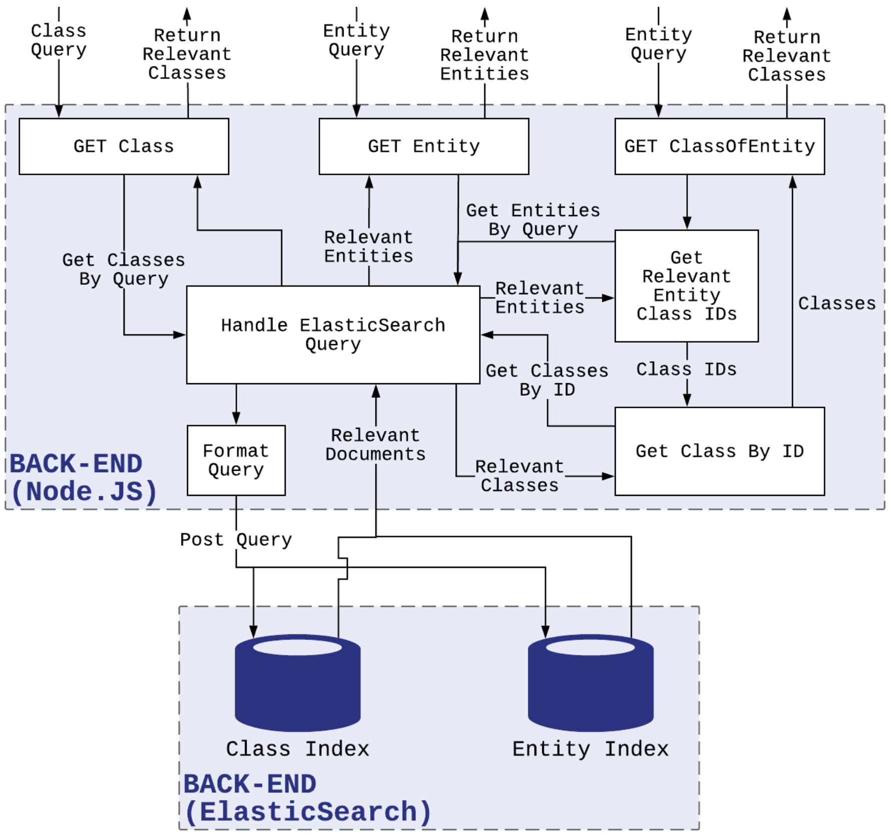


Figure 4.9: An overview of the back-end, its' endpoints and functions.

4.3 Class Retrieval and Ranking

To facilitate a fast and accurate full-text search for class retrieval, a doubly indexed ElasticSearch cluster was set up. The cluster splits the triple store data into *Class* and *Entity* indices, both aimed at retrieving classes. As there are orders of magnitude more entities than there are classes, separating the two improves both the precision and recall of a user class-query. Class-queries also are made quicker, due it consisting of a small subset of the total number of entities.

In cases where there are no results for a given class-query, the entity index is used as a fallback. This fallback approach works by first retrieving a ranked list of relevant entities, extracting their parent class ids, and subsequently the class entities themselves. If a user searches for an entity, they should then be presented with a list of classes with the class of the most relevant entity at the top.

4.3.1 Index structure

Both the class and entity indices share the same document[7] fields. All are indexed by their `label`, `aliases` and `desc`(ription), forming a good foundation for full-text retrieval. Labels and Aliases are also indexed by their *edge n-grams*, enabling search-as-you-type functionality. N-grams are created at index-time and added as an extra field in the document. Each document also has an ID field that allows for direct lookups. This ID is set as an entities unique Wikidata ID, which make *parent-class* lookups fast.

In addition to the textual fields, the class- and entity- documents have numerical fields used in their ranking. Class documents have a `weight` field, which is a normalized score in the range $[1, 2]$ based on the degree of the class. Entity documents have the `parents`, `num_props` `num_links` fields. `parents` contains the IDs of the classes the entity belongs to. `num_props` denotes the number of property-statements applied to an entity. `num_links` is the number of *sitelinks* an entity has, which is the number of wiki-pages (-pedia, -media, -books, etc.) linking to it. Some entities may have Wikipedia pages in many languages, in which case they are all counted.

Class Index At the core of Lister is the class retrieval system. Without it, finding relevant entities would require a text search in the triple-store itself, leaving it to the user to discern class entities from the rest. As full-text search is currently not viable using SPARQL, the choice was made to build an index for class-entity search. The class index is hosted on an ElasticSearch cluster which the front-end can query indirectly.

[7]An entry in an ElasticSearch index is referred to as a document.

The class index is comprised of all class-entities which other entities are members of. It was populated by exploiting the linked data topology of the '`instance of`' property, pruning all entities without an *inbound* `instance of` relationship. To store all of WikiData's $\approx 100,000$ classes with member entities, $\approx$`60MB` of disk space is required. Half of the classes in the class index have three or fewer member entities. Many low degree classes are also of poor quality[8]. Classes with less than five member entities were, therefore, pruned to improve the overall 'quality' of the class set. This leaves an index with $\approx$ 17500 entities, requiring $\approx$`11MB` of disk space. This makes the index cheap to store with a low memory and storage footprint, and fast to search as the number of class entities is relatively low.

By only indexing classes with member entities, a query will *always*[9] return results to the user. Empty pages throw users off their information scent, and creates uncertainty regarding *why* the result set was empty. Ensuring that the user will never receive an empty result set is therefore a *crucial* usability aspect.

Entity Index The second part of the Double Index system consists of the entity index. The entity index is three orders of magnitude larger, requiring $\approx$`15GB` of storage space, and consisting of over 51 million entities. The primary function of this index is to provide a fallback search in the cases where the class index has no relevant results. This is done by returning the ids of entities *parent classes*, which in turn are displayed to the user. Secondary to the class lookup function is the pure entity search, which provides entity matching for the filter function outlined in 4.4.2.

4.3.2 Class Retrieval

Queries are made continuously to the backend as the user types their query, providing a *search-as-you-type* functionality. The received query string is first tokenized using the Unicode Text segmentation algorithm[10], and most punctuation symbols are removed. The tokens are subsequently lowercased, and stemmed using the Lucene 'EnglishPossessiveFilter'[11], removing trailing ''s'.

Documents are first retrieved on whether they contain one or more of the specific tokens of a query. A query can match either the `label`, `aliases`[12], or `desc` fields. A query in progress for 'university' like '`univer`' would likely not match any document, or worse, match an unrelated document. The `label` and `aliases` fields are therefore indexed on their *edge n-grams* as well, providing the prefix matching neccesary for the search-as-you-type feature.

[8]Extremely specific, a duplicate of an existing class, acts of vandalism, or test-classes.

[9]Unless the internet connection is lost or the server fails during querying.

[10]`http://unicode.org/reports/tr29/`

[11]`https://lucene.apache.org/core/8_5_1/analyzers-common/org/apache/lucene/analysis/en/EnglishPossessiveFilter.html`

[12]`aliases` contains a list of alternate labels.

Each token is executed as an independent query, the scores of which are summed up for each document. Furthermore, if a query term is in the ElasticSearch dictionary and has one or more synonyms, the query is expanded to include those. The matching documents are finally ranked according to the functions detailed in 4.3.3.

Example Queries

To demonstrate how the Double Index system works in practice, three use-cases are presented below. These demonstrate the terse, verbose, and entity-based use of the Lister search engine and how the index handles these. In all three cases, the user is interested in retrieving a list of mountains. All cases begin with the user typing their query into the front-end search bar which, via the back end, queries the double index.

Terse Queries The user has understood that the Lister search engine is based around list retrieval of classes. They, therefore, start typing in their desired class to list, "*Mountain*". Queries are made as the user types and return the indexes best estimate based on the *query edge n-gram*. Being a high degree class, `mountain` is the top search result after the user has typed in ‘`mou`’. The user sees their desired class and subsequently selects it, completing the query process. A diagram of this idealized query process is shown in figure 4.10.

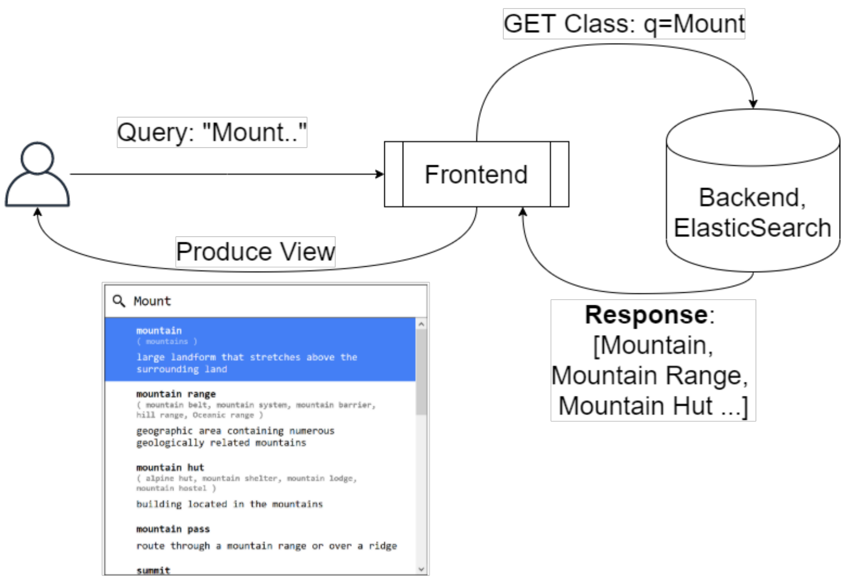
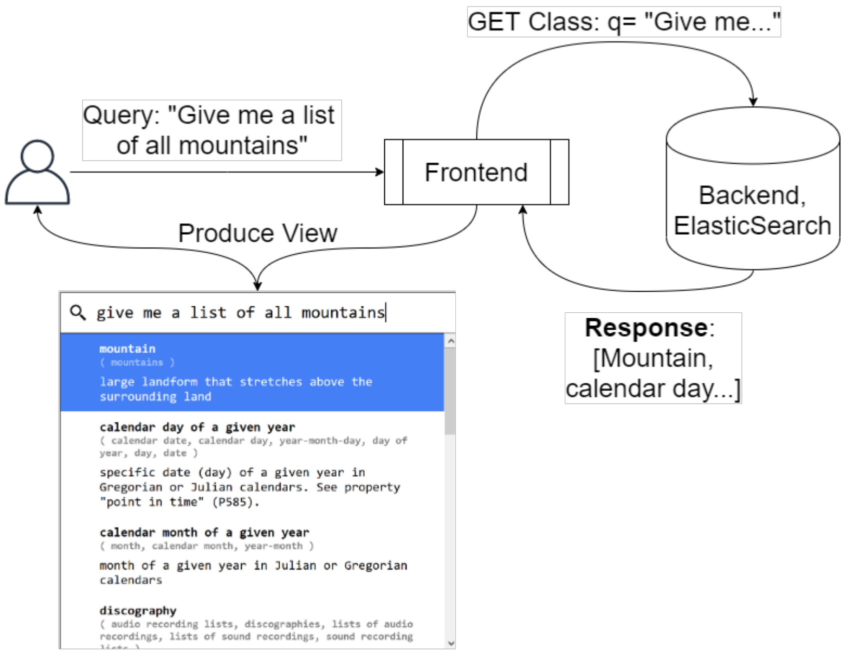

GET Class: q=Mount
Query: "Mount.."
Frontend
Backend, ElasticSearch
Produce View
Response: [Mountain, Mountain Range, Mountain Hut ...]
Mount
mountain
(mountains)
large landform that stretches above the surrounding land
mountain range
(mountain belt, mountain system, mountain barrier, hill range, Oceanic range)
geographic area containing numerous geologically related mountains
mountain hut
(alpine hut, mountain shelter, mountain lodge, mountain hostel)
building located in the mountains
mountain pass
route through a mountain range or over a ridge
summit

Figure 4.10: A query in the process of being formulated returning intermediate results.

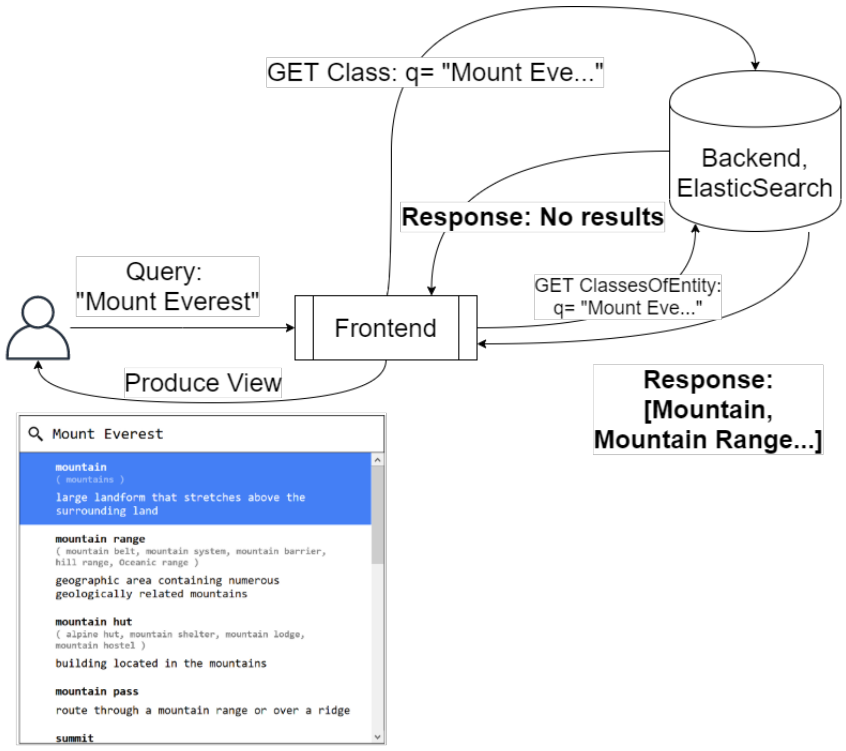
Verbose Queries The user has an idea that the Lister search engine retrieves lists of entities, but prefers to search using sentences than keywords. This user types "`Give me a list of all mountains`" before reviewing their results. The query is broken down into keywords which are all treated as possible edge n-grams and complete search terms. As ‘mountains’ maps to a class with a high weight, its’ result is returned at the top of the result list.


GET Class: q= "Give me..."
Query: "Give me a list of all mountains"
Frontend
Backend, ElasticSearch
Produce View
Response: [Mountain, calendar day...]

```
give me a list of all mountains

mountain
( mountains )
large landform that stretches above the
surrounding land

calendar day of a given year
( calendar date, calendar day, year-month-day, day of
year, day, date )
specific date (day) of a given year in
Gregorian or Julian calendars. See property
"point in time" (P585).

calendar month of a given year
( month, calendar month, year-month )
month of a given year in Julian or Gregorian
calendars

discography
( audio recording lists, discographies, lists of audio
recordings, lists of sound recordings, sound recording
```

Figure 4.11: A typical query, resulting in amore diverse result set.

Entity-based Queries The user searches for an entity instead of a class. Instead of being presented with an empty result list, Lister tries to map their entity-query to a relevant class. Should the class index return no results, the `ClassesOfEntity` endpoint is queried as a fallback. This function retrieves and returns the classes of the top ten entities related to a query. These are presented to the user as shown in figure 4.12.


GET Class: q= "Mount Eve..."
Backend, ElasticSearch
Response: No results
Query: "Mount Everest"
Frontend
GET ClassesOfEntity: q= "Mount Eve..."
Produce View
Response: [Mountain, Mountain Range...]

Mount Everest

mountain
(mountains)
large landform that stretches above the surrounding land

mountain range
(mountain belt, mountain system, mountain barrier, hill range, Oceanic range)
geographic area containing numerous geologically related mountains

mountain hut
(alpine hut, mountain shelter, mountain lodge, mountain hostel)
building located in the mountains

mountain pass
route through a mountain range or over a ridge

summit

Figure 4.12: A query with no class hit, falling back to a class-of-entity search.

4.3.3 Result Ranking

Scoring Mechanism ElasticSearch uses the Okapi BM25 ranking function for calculating the base score of a document. This ranking may be used as-is or built upon using ElasticSearch scoring functions. On the Lister cluster, the base score calculated independently for multiple fields of a document. In this instance the `label` and `label_ngram`, `alias` and `alias_ngram`, and `desc` fields. When a multi-field query is submitted, ElasticSearch by default scores each document with the `best_fields` function. `best_fields` takes the maximum score of a documents field, returning that as the score for the document as a whole. Fields can have multipliers applied, increasing or decreasing their scores. Both the class and entity indices have their `label` and `alias` field scores multiplied by 3 and 1.5 respectively, ensuring that title hits are weighed heavier than the description or `ngram` hits. This is hereafter referred to as the *multi-field* ranking function.

Class Ranking The ranking function $S(c)$ was defined as detailed in function 4.1 to improve upon the default class-search result ranking by ElasticSearch.

There are a large number of classes, and several of them have a high *lexical similarity*[13]. In order to distinguish lexically similar classes and improve the mean reciprocal rank, scores are weighted based on *importance*. This importance score is a combination of the weight of class c, $W(c)$, the number of properties a class has $n_p(c)$, and the number of pages that link to that class $n_l(c)$. This results in the score $S(c)$;

$$S(c) = ES(c) \times W(c) \times \log((n_p(c) + 1) \times 2^{n_l(c)}) \tag{4.1}$$

The function $ES(c)$ is the score calculated by the weighted ElasticSearch multi-field scoring function.

The weight $W(c)$ is based on the degree of a class. The degree cannot be used directly, as the largest degrees are in the millions, while the median is 3. To reflect that the difference between a degree of 2 or 3 million entities is small, but the difference between 1 and 100 large, the logarithm of each degree is taken. These scores are then min-max scaled between 1 and 2, making it a positive factor in the scoring equation $S(c)$.

The number of properties and site links of a class are used as heuristics for importance. Site link count was used as the most significant factor of importance, making classes with more inbound website links higher scoring. The number of properties a class has is also accounted for, making 'detailed' classes more important. However, classes exist that has a high number of properties with a likely low relevance to a user query[14]. The number of links is therefore valued significantly more than the number of properties, though properties also boost scores.

Entity Ranking To improve the default ranking of entities $ES(e)$, the function $S(e)$ was defined. $S(e)$ was constructed such that given a list of entities with the same *lexical* similarity to a query, $ES(e)$; they should first be ordered by the number of site links, then the number of properties. This is achieved by scaling the scores with the function detailed in equation 4.2, which is computed at search-time.

$$S(e) = ES(e) \times \sqrt{(n_p(e) + 1) \times (n_l(e) + 1)} \tag{4.2}$$

The function $ES(c)$ is the score calculated by the weighted ElasticSearch multi-field scoring function. This score is thereafter scaled by a function of the number of properties and site links an instance has.

[13]Title, label, or description that is similar in terms of their content

[14]Speculation based on these classes generally having very few member instances (< 10), and belonging to a very specific field.

4.4 Data Interaction

The data table has a number of features aiding the user in interacting with and exploring their data. Several steps have been taken to maintain visibility, legibility, and accessibility. These are highlighted in 4.4.1. By exploiting data topology functions like adding and removing information and adding 'native SPARQL' filters is possible. These are highlighted in 4.4.2.

4.4.1 The Data Table

The following is a non-comprehensive list of the various features of the data table. These, and other features, ensure that the user can easily and painlessly view and interact with their data.

Legibility Visual clutter is kept at a minimum by removing all row- and column line delimiters, and minimizing the use of colours. All columns share the same width, and all rows the same height. Classic zebra-striping is eschewed in favour of highlighting rows on mouseover, ensuring that reading data row-wise is easy. Column-, row-, and cell-padding and margins follow the Material Design[15] guidelines for maintaining legibility. Numerical and temporal columns are right-aligned, and temporal columns all follow the `dd.mm.yyyy` format. These features make numerical and temporal data easy to compare at a glance. Finally, each column header features an *order-by* button. This button allows the user to order the table data by any column desired, based on the data type of said column. The button reflects the table ordering by changing its' icon corresponding to whether the column is ordered ascending, descending, or unordered.

Sticky Headers Scrolling may be required to take in all the content on a wide or long data table, which in turn moves the entity-column or table headers out of view. Losing the context of what entity or column the user is currently viewing can make parsing information from the table harder. This problem is solved by *sticking* the row-header to the top of the page, keeping it visible as the user scrolls down. The entity column on the far left also sticks so that users can see which entities belong to which properties on wide tables.

Placeholders After selecting a class, the user is presented with the results page. Lister executes a SPARQL-query to an endpoint in the background, retrieving relevant triples. While retrieving, animated placeholder (fig 4.13) boxes indicate to the user that their data is on its way. Retrieval-time varies based on the total number of entities belonging to the selected class. For large classes, placeholders are therefore a necessary clue to the user that the system is functioning as intended.

[15]`https://material.io/components/data-tables`

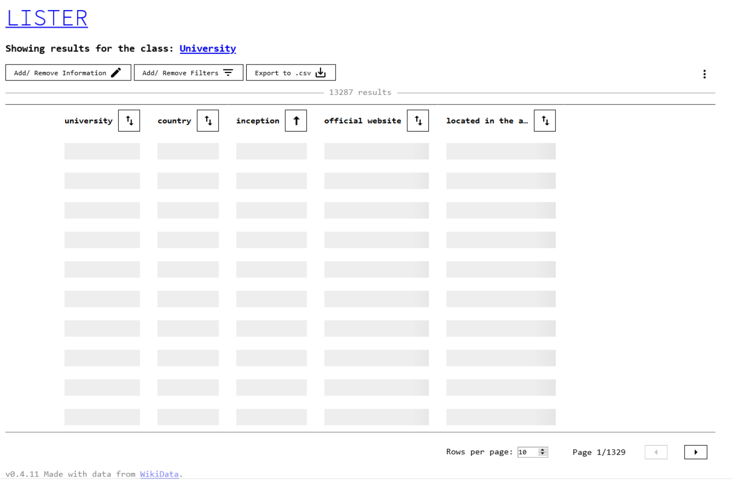


Figure 4.13: Animated placeholder boxes indicate that the program is retrieving information.

URIs URIs are at the heart of the Semantic Web and are a central component in an RDF triple store. The Lister system facilitates data exploration by including URIs where they are available, displaying them as standard web-links. By allowing the user to follow links to their respective sites, they might discover more information about the selected class or retrieved entities. This can also help the user disambiguate their selected class, should their results look different than expected.

Pagination Many classes have thousands of member entities, with some having millions. Retrieving large numbers of triples will take a toll on client computer memory, bandwidth, and retrieval time. Results sets are therefore paginated by default, with a default limit set at 10 items. Clicking the next- or previous button increments or decrements an offset applied to the result set. Counting the total number of items in a result set is a relatively fast operation, typically taking less than 30 seconds even for millions of items. This makes it possible to indicate to the user how large the result set is, while still only displaying a fraction of it. Having the ground truth total number of queries also allows for a page-counter, showing the user where in the result set they are.

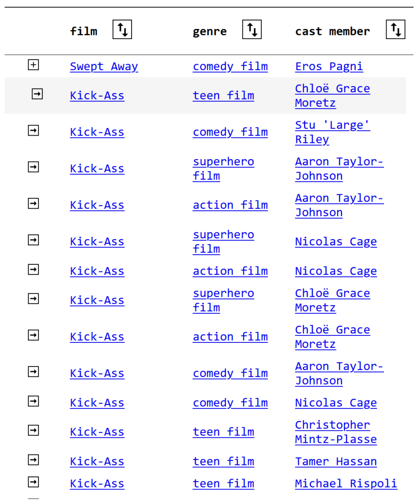
Auto-Grouping Many entities have more than one row in a result set. This is due to an entity having more than one value for a given property, all combinations of which being equally true. An example of this can be seen in figure 4.14 The number of rows representing an entity is, therefore, the product of the number of values said entity has per property. An equation detailing this is provided in 4.4.

Let C be the set of all properties selected in a table, and c the index of a property of that set. $p_c(e)$ is then set of values the entity e has for property number p_c as shown in equation 4.3. The number of rows of that entity, $r(e)$, is then the product of the cardinalitites of e for all properties p_c.

$$p_c(e) = \{v_1(e), v_2(e), \dots, v_n(e)\} \tag{4.3}$$

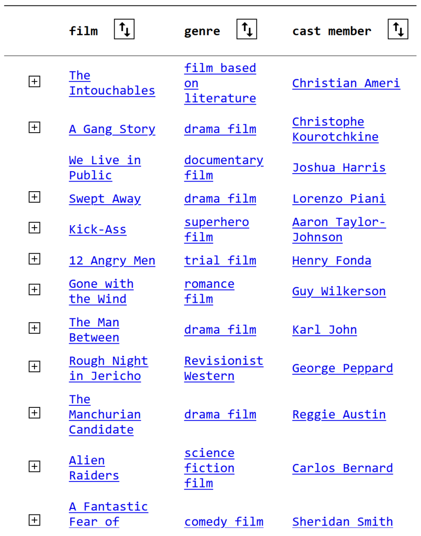
$$r(e) = \prod_{c=1}^{|C|} |p_c(e)| \tag{4.4}$$

These apparent duplicates clutter the result set, not letting the user see the actual diversity of retrieved entities. A group-by operator was therefore added, and activated by default. An arrow symbol to the left of a column indicates to a user that a row is expandable. Upon clicking said row, all values of it are displayed, after which one value may be selected for display by clicking on it. This allows the user to both dig deep into specific entities, but also to see that the result consists of several other entities as well.



	film ⇅	genre ⇅	cast member ⇅
⊞	Swept Away	comedy film	Eros Pagni
⇥	Kick-Ass	teen film	Chloë Grace Moretz
⇥	Kick-Ass	comedy film	Stu 'Large' Riley
⇥	Kick-Ass	superhero film	Aaron Taylor-Johnson
⇥	Kick-Ass	action film	Aaron Taylor-Johnson
⇥	Kick-Ass	superhero film	Nicolas Cage
⇥	Kick-Ass	action film	Nicolas Cage
⇥	Kick-Ass	superhero film	Chloë Grace Moretz
⇥	Kick-Ass	action film	Chloë Grace Moretz
⇥	Kick-Ass	comedy film	Aaron Taylor-Johnson
⇥	Kick-Ass	comedy film	Nicolas Cage
⇥	Kick-Ass	teen film	Christopher Mintz-Plasse
⇥	Kick-Ass	teen film	Tamer Hassan
⇥	Kick-Ass	teen film	Michael Rispoli

(a) Multiple statements for the same film, one for each combination, with the option to select one.



	film ⇅	genre ⇅	cast member ⇅
⊞	The Intouchables	film based on literature	Christian Ameri
⊞	A Gang Story	drama film	Christophe Kourotchkine
	We Live in Public	documentary film	Joshua Harris
⊞	Swept Away	drama film	Lorenzo Piani
⊞	Kick-Ass	superhero film	Aaron Taylor-Johnson
⊞	12 Angry Men	trial film	Henry Fonda
⊞	Gone with the Wind	romance film	Guy Wilkerson
⊞	The Man Between	drama film	Karl John
⊞	Rough Night in Jericho	Revisionist Western	George Peppard
⊞	The Manchurian Candidate	drama film	Reggie Austin
⊞	Alien Raiders	science fiction film	Carlos Bernard
⊞	A Fantastic Fear of	comedy film	Sheridan Smith

(b) The same result set, with one value for Kick-Ass having been selected.

Figure 4.14: An example showing entities with several multi-valued properties, and how grouping these can produce a more diverse result-set.

Downloading Data In addition to exploring data online, users can download their data. Lister collects and the data with the desired columns, group selections, and filters. What the user sees is what they get. The data is formatted as a comma separated values (`csv`) file, which can be painlessly imported into mainstream spreadsheet applications and is a standard open format for data-science applications. Example use cases for downloaded data include combining the Lister data with a local source, integrating with a presentation tool, or for use in a data-based project.

Responsive Design Modern web agents are more than full-sized browser windows on desktop computers, and according to statcounter[16], the most common screen resolution online is $360x640$ pixels. In order to support smaller screens and browser windows, responsive design has been implemented. Responsiveness means that the content scales to fit the viewport of the user web browser, and ensures a functional design for screens of all sizes. Ease of use and accessibility concerns for touch devices has also been addressed by using semantically tagged input elements. These inputs use the browsers own input functions, which are optimized for input type (*text, time, numerical*) and I/O devices such as the mouse or touch display.

4.4.2 Exploiting Linked Data Topolgy

Adding and Removing Properties When exploring new data, users might not know what properties are relevant to their selected class. Properties that might seem relevant may not be, and properties thought to be irrelevant might be widely used. Finding the right properties is, therefore, a matter of trial and error, which may be time-consuming on large result sets. This issue is solved by *pre-fetching* the properties relevant to the selected class.

The most common properties are found using the query detailed in the listing below. `${var}` denotes a variable field, supplied by the frontend. The query gets entities of a class, which it then groups and counts the number of properties of, returning a ranked list of the most popular properties. This ensures that all properties are relevant for members of the selected class.

```
  SELECT ?uri ?name ?desc (COUNT(distinct ?p) as ?count) {
    {
      select ?p {
        ?p wdt:P31 wd:${entity}.
      } limit ${limit}
    }
    ?p ?a ?val.
    ?uri wikibase:directClaim ?a .
    ?uri rdfs:label ?name.  filter(lang(?name) = "en").
    ?uri schema:description ?desc.  filter(lang(?desc) = "en").
} group by ?uri ?name ?desc order by DESC(?count)
```

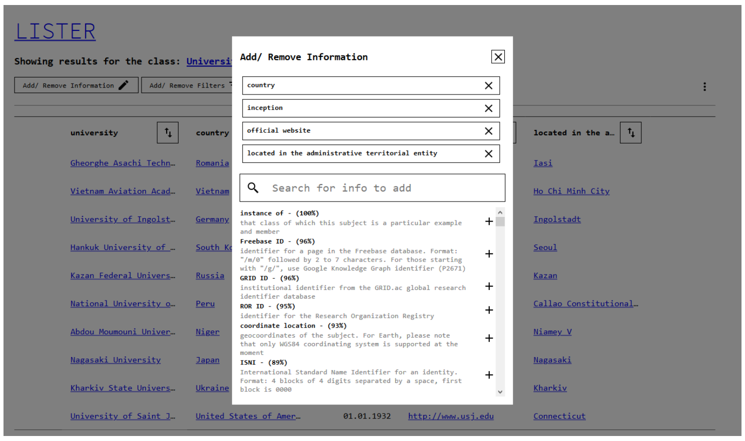
[16]`https://gs.statcounter.com/screen-resolution-stats` retrieved 16.05.20

To give users an immediate feel for the class they selected, and an information scent, the *four most common properties* are already added when the initial results are displayed. This query should resolve quickly, so the user is not left waiting longer than necessary. Also, since the most *popular* properties should be applied to most entities, this only counts the properties of 20 entities.

After the initial count, a larger query is made. This gathers and counts the properties for the first thousand entities of a class. The properties are stored in memory and can be added using the 'Add/ Remove Information' button. This opens the modal shown in figure 4.15, displaying the available properties along with their frequency.

The large query is made in the background, blocking the 'Add/ Remove Information' button in progress. In cases where the number of entities is less than this, all properties are retrieved. In other cases, there is a risk of some relevant properties not being included. It is assumed here that 1000 entities are a representative sample, however, and therefore assumed that properties not appearing in that subset are likely irrelevant.

With these convenient functions, data exploration is made more accessible by removing 'dead-end properties' and ranking relevant ones by frequency. The user may also get a better idea of the members of a class by which 'company' it keeps, determined by the most common properties that are automatically added.



Add/ Remove Information

country
inception
official website
located in the administrative territorial entity

Search for info to add

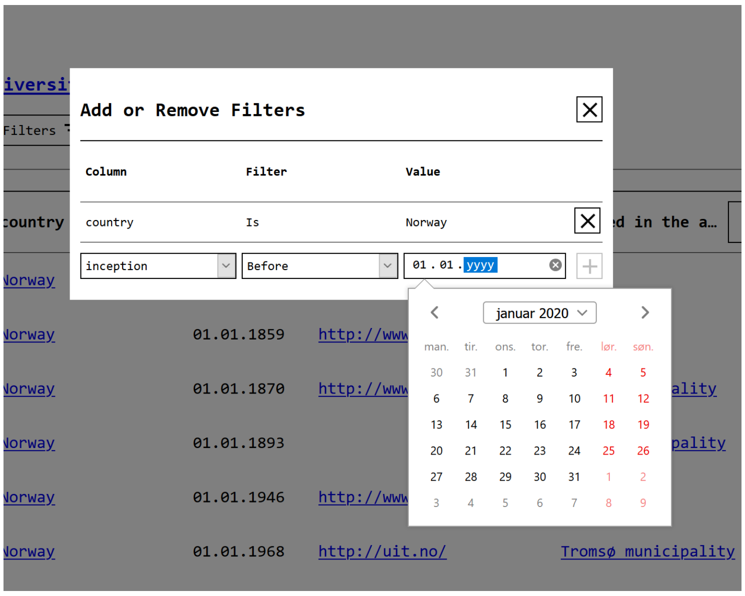
instance of - (100%)
that class of which this subject is a particular example and member
Freebase ID - (96%)
identifier for a page in the Freebase database. Format: "/m/0" followed by 2 to 7 characters. For those starting with "/g/", use Google Knowledge Graph identifier (P2671)
GRID ID - (96%)
institutional identifier from the GRID.ac global research identifier database
ROR ID - (95%)
identifier for the Research Organization Registry
coordinate location - (93%)
geocoordinates of the subject. For Earth, please note that only WGS84 coordinating system is supported at the moment
ISNI - (89%)
International Standard Name Identifier for an identity. Format: 4 blocks of 4 digits separated by a space, first block is 0000

university	country			located in the a…
Gheorghe Asachi Techn…	Romania			Iasi
Vietnam Aviation Acad…	Vietnam			Ho Chi Minh City
University of Ingolst…	Germany			Ingolstadt
Hankuk University of …	South Ko			Seoul
Kazan Federal Univers…	Russia			Kazan
National University o…	Peru			Callao Constitutional…
Abdou Moumouni Univer…	Niger			Niamey V
Nagasaki University	Japan			Nagasaki
Kharkiv State Univers…	Ukraine			Kharkiv
University of Saint J…	United States of Amer…	01.01.1932	http://www.usj.edu	Connecticut

Figure 4.15: The Add or Remove Information modal.

Filtering Data If a user has a specific information need beyond just adding properties, filtering comes into play. This allows the user to perform standard filtering tasks, depending on the data type of the column they have selected. The dialogue shows a list of currently applied filters at the top, with a menu for adding filters at the bottom. Filters are built stepwise by first selecting a column, then a filter, then a value for that filter. An example filter dialogue is shown in figure 4.16. A comprehensive list of the available filters can be seen below. All except the URI matching filter applies standard SPARQL filter statements to the background query.

- **URI**: Matches (Graph matching on URI), Starts With, Is Not
- **String**: Matches (String equals), Starts With, Is Not
- **Numerical**: Larger Than, Smaller Than, Is Not, Equals
- **Temporal**: Before, After
- **All**: Remove missing values



Add or Remove Filters

Column	Filter	Value
country	Is	Norway
inception	Before	01 . 01 . yyyy

Figure 4.16: The Filter Columns modal while adding a temporal filter.

4.5 Chosen Technologies

4.5.1 Front-End

When choosing front end technology, several factors were considered. The application is meant to be highly portable and able to scale to a large number of users. To achieve this, a web-app architecture was chosen to place most, if not all, of the code execution client-side. There are numerous languages and frameworks in which such a system could be implemented, out of which React.js was chosen. The React frameworks were selected for its familiarity, popularity, and large package ecosystem.

React.js

React.js[17] is a JavaScript library for creating web applications. It is a code-first approach to web development where JavaScript is written to "serve" HTML. This is done by writing code that provides the Document Object Model (DOM) with JavaScript XML, which specifies the HTML to be rendered. React maintains a Virtual DOM in-memory that monitors any changes to be made to the DOM. In the event of a change, only the altered element will be updated in the actual DOM. This makes re-rendering only the changed components on the web-page possible with no extra effort from the developer.

create-react-app Create-react-app is an installer script made for simplifying the set-up process when starting a new React project. In a chosen directory it will set up configuration files, add all necessary dependencies, and sets up scripts for starting the project locally and building it. CRA was used for set-up in this project to simplify starting a new project.

4.5.2 Back-End

Node.js

Node.js[18] is a JavaScript runtime environment that allows developers to use JavaScript outside of a web browser. This means that practically the same programming language can be used both front- *and* back-end. Libraries that work with JavaScript are also compatible with Node.js. Endpoints over HTTP can be set up using native methods, or using a framework like Express.js[19] as was used in this project.

ElasticSearch

Setting up the back-end query API required a simple text search engine with the ability to tweak search and ranking parameters and fast processing times. Elast-

[17]https://reactjs.org/
[18]https://nodejs.org/
[19]https://expressjs.com/

icSearch[20] is a Lucene-based, NoSQL, schema-free document store with a focus on fast retrieval of data. The search service provides JSON results over HTTP by a RESTful API, making it easy to integrate with web services. The API provides the user with a full set of functionalities, from index creation and specification, querying, and admin tasks.

Kibana

Kibana[21] is a dashboard for managing ElasticSearch clusters. The dashboard can be used to visualize and query data in the ElasticSearch indices, set up and manage user privileges, and handling logging and metrics tasks. It runs as a local web service, accessible with a web browser.

4.5.3 Data Processing

In order to set up the ElasticSearch indices detailed in 4.3.3, some data-processing was necessary. For this task, Python 3 was chosen. Python is widely used in data-processing and science contexts, and through packages like Pandas[22] and Jupyter Notebook[23] data exploration is simplified.

All data used was sourced from Wikidata. Wikidata has, in addition to hosting SPARQL endpoints for data retrieval, made their entire database available for download[24]. The data dump comes in a variety of formats, of which the JSON dump was chosen for being particularly easy to load and modify using the standard python package `json`.

Jupyter Notebook

Jupyter Notebook is an interactive web-based environment in which Python code can be executed iteratively. It runs on a locally hosted web server, which the developer interacts with through a web-browser of choice. Jupyter is based around Notebooks, which consist of blocks of code and markup that are executed in a stepwise manner. Variables are stored until the notebook is restarted, or the variable is deleted or re-assigned. This makes data exploration simple, as there is no need to re-run a long processing pipeline in order to solve an issue with a single step. It is also an excellent environment for one-off tasks - usually of the *Extract Transform Load* variety. As Jupyter allows for both Python and Markdown code blocks, creating self-documenting code and stepwise instructions is easy.

[20]`https://www.elastic.co/`
[21]`https://www.elastic.co/kibana`
[22]`https://pandas.pydata.org/`
[23]`https://jupyter.org/`
[24]`https://www.wikidata.org/wiki/Wikidata:Database_download`

DASK

Pandas is a widely used package for data exploration and transformation through its' DataFrame structure. DataFrames and the data loaded into them are kept entirely in-memory, making them fast but also limiting their size to available RAM. DASK[25] is a Python package which enables, among others, Pandas DataFrame-like structures while keeping the data on-disk. DASK provides lazy execution of code and operations on subsets of data, making it excellent both for ETL operations and fast exploratory analysis. Due to the size of the Wikidata data dumps - around `55Gb` gzipped and `1100Gb` unzipped - DASK was an ideal candidate. The DASK Bag and DataFrame API's were therefore extensively used to process the data, preparing it for loading into the ElasticSearch indices.

[25] `https://dask.org/`

Chapter 5

Results and Discussion

5.1 User Tests

5.1.1 Test Population

Subjects for the user test were chosen based on their availability and relevance as a potential end-user. Subjects were recruited from the authors' friends, colleagues in academia, and journalists from the research magazine Apollon[1]. An overview of all test subjects can be seen in table 5.1.

Test Number	Subject Number	Relevance to the Project
1	1	Professor, Computer Science
	2	Computer Science Student
	3	UI/UX Designer
	4	Journalist
	5	Journalist
2	6	Decision Maker
	7	Computer Science Student
	8	Journalist
	9	Professor, Computer Science
	10	Programmer

Table 5.1: Table of user test participants.

5.1.2 Results

First Iteration

The results of the first round of user-testing highlighted many issues with the original prototype. An overview of positive and negative remarks can be seen in table 5.2.

[1] `https://www.apollon.uio.no/`

Positive Remarks:	Negative Remarks
+ Clean design	- It is hard to find the correct class in the result list
+ Intuitive search interface	- The wording used does not make sense to me
	- I'm not sure if the program is loading or has stopped working
	- "Enter" keypress selects class instead of initiating search
	- A lot of bugs

Table 5.2: Some common paraphrased remarks from user-testing the first prototype.

Several test users reported problems with using the search bar to find relevant classes. These users either selected a class with no member entities or got so many irrelevant results that it was hard to pick out the right one. Users also intuitively searched as if they were using a regular search engine, using sentence structures like 'Give me all (...)' or 'List of all (...)'. Some users also queried for entities of the class itself, such as querying for 'NTNU' for finding the class 'University'. It was therefore made apparent that the class search functionality had to be robust. The class search furthermore had to provide a fitting mapping for queries that are not directly related to the desired class but can be found in the 'neighbourhood'.

The table interface also had problems, most notably when it came to wording. "Columns" were used to denote properties applying to "entities", which are the members of the class the user selected. The users appeared to struggle to build a mental model as to how exactly columns related to entities, and what exactly entities were.

Finally, there were substantial performance issues. Some were rooted in the haphazard data-structures of the initial prototype, but most to the retrieval of data from the triple-store endpoint. The tool retrieved almost all information from the endpoint, which included some very taxing queries on what properties were available, what values the different properties could have, and a too-large limit for retrieving instances altogether. Among others, the following improvements were made after the first iteration:

- A double index was built, providing fast and accurate class search & retrieval
- All 'empty' classes were pruned
- Loading animations indicating that the page is fetching data
- Improved wording
- Extensive rework of the codebase to eliminate bugs
- Automatic data-type detection, so that filters can be customized to specific data types.

Second Iteration

The results of the second round of user testing validated that the prototype was 'headed in the right direction'. An overview of positive and negative remarks can be seen in table 5.3.

Positive Remarks:	Negative Remarks
+ The Class Search paradigm is easy to grasp after the first search	- It is hard to know what to search for initially
+ Clean Design	- The search "limit" is confusing; Total number of items or restriction?
+ Feels Powerful	- I'm not sure if the program is loading or has stopped working
	- The wording used does not make sense to me

Table 5.3: Some common paraphrased remarks from user-testing the second prototype.

By improving the class search, the mental models of users seemed to align better with the class-first-approach of the search engine. By seeing only classes that contained members, and having ranked 'likely relevant' classes highly, the users understood that they were supposed to choose a class instead of an entity. Even though users queried in the same way as observed in the first iteration, the accuracy of their results improved dramatically. The users could now query in their preferred way, and still get results containing relevant classes. Much of this could be attributed to the removal of stop words, simple passive stemming and exploiting graph topology when ranking results.

The Material Design guidelines for data tables[2] were implemented in the table view. By also reducing the number of results on one page, the *visibility* of the data improved substantially. Users reported that this made the results view alltogether feel more orderly, and in turn more usable. Among others, the following improvements were made after the second iteration:

- Improved Class Search ranking algorithms
- Moved search settings to a 'three dot menu', simplifying the UI
- Grouping of rows with the same entities to maintain diversity
- Query caching makes the site feel less sluggish, saves network bandwidth
- Implemented Material Design guidelines
- Added Search Result page for displaying results from Class Search in a separate page.
- Enable exporting of data to CSV format for use with spreadsheet applications

[2]`https://material.io/components/data-tables`

5.1.3 Feedback on the Final Prototype

As part of the System Usability Scale tests detailed in 5.2, users were allowed to enter qualitative remarks and feedback on the design. These results were also gathered into a table, which can be seen in 5.4:

Positive Remarks:	Negative Remarks
+ Novel idea and application for search	- The wording used does not make sense to me
+ Clean Design	- Does not scale well for smart-phone use
+ Feels Powerful	- Too monotonic and messy design
+ Great for research	- The filter function is awkard

Table 5.4: Some common paraphrased remarks from user-testing the final prototype.

The problem of wording remains even in the final prototype. Attempts were made after each iteration to formulate the text in the prototype better. Especially the wording for adding or removing properties and the names of different kinds of filters proved to resonate poorly with users. While users struggled to come up with better names, many suggested small helper texts to provide context on the different functions.

Users had several remarks on the experience using the filter functionality. Users expected filters to be applied as they were filled in. This meant that having to click the '+' sign or press the enter key to set a filter was found to be confusing. A number of users also missed the ability to edit their filters.

Some users remarked that the prototype did not scale well for phones, which have smaller, vertical displays. As the prototype was only tested using a large desktop monitor, this issue had not been caught early in the process. It is therefore understandable that users found the search engine poorly designed for mobile devices.

Finally, there seemed to be two opposing views on the quality of the design. Users reported both that it was messy and clean, functional and oversimplified. Quality of design is in large part a subjective rating, where tastes differ. Future design work should focus on creating a more aesthetic and holistic interface.

5.2 System Usability Scale

5.2.1 Test Population

The call to action for testing was posted in various online communities. In order to gather respondents with little to no experience using structured query languages, a post was made on the author's Facebook page. To mitigate possible responder bias, the post was also shared to a large group ($\approx$ 3000 members) of developers called 'Kode24-klubben'. To gather respondents with experience in structured query languages, the call to action was also posted in Wikidata's community chat and weekly newsletter.

The test population was divided into three groups of people grouped by their previous experience using structured query languages. The working hypothesis was that users familiar with structured data querying languages would quickly grasp the tool, and bias the results in a positive direction. Part of the motivation for creating this tool was also to help the general public utilize semantic data; Casual users usability scores should therefore be given special consideration. The distribution of participants over these three groups can be seen in table 5.5.

Experience With Structured Languages	n	% of Total
None	18	$\approx$ 21%
A Little	37	$\approx$ 43%
Experienced	31	$\approx$ 36%
Total	86	100%

Table 5.5: Test participants as grouped by their experience level in structured query languages

5.2.2 Results

The collection of calculated usability scores can be seen visualized as a bar chart in figure 5.1, and some descriptive statistics by user group in 5.6.

Results from users that did not successfully complete the test were pruned from the working data. Several of the users did not finish the test due to a bug making the tasks impossible for Safari users, and five others reported that they did not understand the tasks but still rated the system. These are artefacts of the test format, and does not directly reflect the usability of the system.

Due to an error in the data collection form, the first 20 results are missing the score for question 8: 'I found the system very cumbersome/ awkward to use.'. These missing values were substituted with the median of the remaining 63 scores.

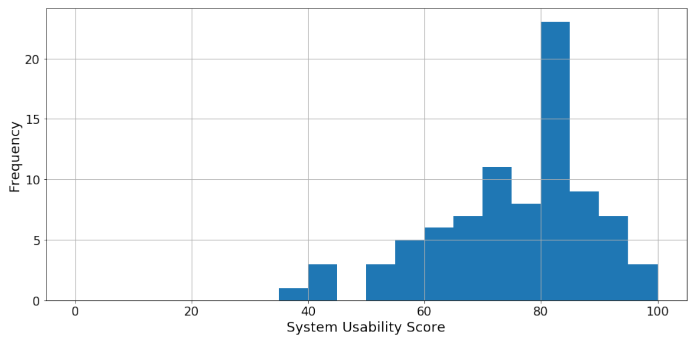


Figure 5.1: The distibution of scores on the System Usability Scale. $n = 86$.

Experience	n	Median	Mean	SD
None	18	80.0	81.11	11.38
A Little	37	77.5	74.86	12.46
Experienced	31	75.0	70.08	14.84
Total	**86**	**77.5**	**74.45**	**13.63**

Table 5.6: The median, mean scores, and standard deviations of usability scores by user experience with structured languages.

5.2.3 Comparison To Similar Systems

Comparison to Web Interface Average Bangor et al. report in their 2008 study that the average usability for web user interfaces is 68.05. A right-tailed one-sample t-test is conducted to see whether or not our usability test results exceed this benchmark. The Lister SUS distribution of scores has the following characteristics: $n = 86$, $\mu = 74.45$, $\sigma = 13.63$. The p-value for this test is $p < 0.0001$, indicating an extremely significant result. This evidence that Lister has a SUS score above the average score for web interfaces.

Comparison to Semantic Web Interfaces Lister can be viewed in comparison with the SUS scores gathered in the literature review in 2.5.2 in table 5.7. The scores of other tools reported in table 5.7 are for 'casual' users. The mean SUS score of the combined user groups with little to no experience using structured languages are considered 'casual' end users in this study. The average SUS score for casual users is, therefore included as Lister (casual) to provide an equal basis of comparison.

Tool	**SUS Score**	**Tool Type**
Affective Graphs	55.00	QBI (Graph)
Ginseng	55.10	NLI
K-Search	41.25	QBI (Form)
Lister (Total)	74.45	Hybrid
Lister (Casual)	**76.91**	**Hybrid**
NLP-Reduce	56.72	NLI
PepeSearch	75.30	QBI (Form)
Querix	75.73	NLI
Semantic Crystal	61.25	QBI (Graph)

Table 5.7: Table of SUS Scores of surveyed tools including Lister. Tools with more than one reported score have had their highest score included. Lister evaluated by casual users has the highest overall SUS score, and is therefore highlighted.

5.3 Discussion

5.3.1 Findings of this Study

After collecting the results from the user- and SUS-testing, the hypotheses as defined in chapter 1.2 were re-examined. Qualitative results and field observations substantiate the claims made in hypotheses 1-4. The quantitative results gathered from the SUS user-test support hypothesis 5, but it cannot be said for certain that iterative user-testing was the sole factor making the search engine more usable than average. A structured review of how the findings of this study relate to- and support the hypotheses follow;

H. 1 *Full-text search, continuous feedback, and iterative query specification will ensure that query formulation is easy.*

Findings User-tests and qualitative feedback from the online SUS test show that users found query formulation for class-search easy. While few users searched for classes in their initial query, it was observed that users quickly reacted to feedback from the system and adopted more 'class-centric' query formulations.

H. 2 *By displaying data in a fully interactive data-table, users may explore their data at a glance. They may also drill down by filtering, ranking, or following a* URI.

Findings User-testing results indicated that users found the table-interface suitable for interacting with- and exploring their data.

H. 3 *Query result ranking can be improved by scoring results based on the* degree *of the retireved results.*

Findings Result relevance is dramatically improved by using the double-index approach. User-testing found that users received the results they expected querying in their natural language. However, there are still many edge cases where the current system of ranking falls short, indicating a need for an improved ranking function.

H. 4 *Data exploration will be simplified by exploiting data topology to fetch relevant information about the chosen subject, and add human-readable labels to entity URI's.*

Findings User testing indicated that adding common properties by default and ranking properties by frequency aided users in data exploration. Users promptly found the properties relevant to their case and *uncovered previously unknown properties that were of interest*. Users instinctively grasped that the URI 'links' led to more information about that specific entity, indicating a useful metaphor for URI's, and that entity exploration was facilitated.

H. 5 *By performing iterative development with continuous user testing, a better than average SUS result can be achieved.*

Findings SUS test results showed that a significantly better than average user interface was developed. This can, in part, be attributed to the user-testing where several usability issues were found and addressed, and new features suggested and subsequently implemented. Furthermore, the SUS scores for casual users was the highest among those tested indicating improvement over several existing semantic web user interfaces.

5.3.2 Contributions

Addressing the usability issues outlined in chapter 2.5.2, Lister was developed. Lister is a user-friendly semantic search engine utilizing a hybrid combination of natural language querying and query building search specification.

With Lister, the vast amounts of data stored on the semantic web are accessible without knowledge of the underlying data structure, or structured query languages like SPARQL. This democratizes semantic web information retrieval, letting a wider audience reap the benefits of linked data; Users can gather data relevant to *their* information need without relying on a potentially costly third-party data provider. Crowd-sourced triple-stores like Wikidata provides vast amounts of up-to-date data available on almost every imaginable subject. This allows researchers, journalists, and developers to build and download datasets that are ready to integrate into a data-driven project or application, free of charge. Alternatively, a casual user may just want to see what historic sites or attractions are worth seeing in the city they are visiting.

Lister serves as a proof-of-concept on how casual users might draw advantage of the data stored in triple-stores like WikiData or DBpedia. This concept could, in addition to being a stand-alone web service, easily be integrated into contemporary internet search engines like Google. Google already attempts to disambiguate a user query, serving as the ‘class-search’ part of the Lister search engine. With a class identified, the list of member instances could be displayed in a small widget allowing the user to explore their data further.

Furthermore, Lister demonstrates a hybrid approach to semantic query formulation. By combining the simplicity of formulating a natural language query with the ease of use and transparency of query builders, a more user-friendly interface was built.

Finally, Lister demonstrates the necessity for involving end-users in the design and development process. Though not all usability issues were caught before the final usability-test, it is safe to say that Lister would suffer severely from a lack of testing. As noted by Hachey et al. in their survey [7] and in the background chapter of this paper, most semantic web user interfaces lack proper usability testing. Combined with the sub-average SUS scores of those interfaces, this substantiates that future efforts in semantic web interfaces should have an increased focus on usability and user testing.

5.3.3 Validity and Reliability of Results

User Tests

There are a few threats to the validity of user test results. As a number of test users were drawn from a computer science environment, there is a chance that they are better than average at using computer tools. This, in turn, might lead to usability issues for laypeople go unnoticed, or that incorporating feedback from a computer science expert might make a system *less* usable for laypeople. This can and was counteracted by including a number of people without a computer-science related background. Furthermore, many usability issues have little to do with how much or little computer science knowledge the user has. Bugs and poor design are indiscriminate and should be noticeable when overseeing a test subject using the system. Finally, a test subject with a computer science background might also be more sensitive to bad design choices, as they themselves are aware of common pitfalls.

The test subject might report that the system is better than it actually is to please the facilitator. The user might find it uncomfortable to criticize the hard work of another person, or feel that they are not in a position to judge. To counteract this, the facilitator should always tell the test subject that they have not been involved with the design. Observing how the user interacts is, however, the "gold standard", as it should be easy to identify whether or not a user struggles with the design or task at hand.

The test subject might behave differently than if they were in a non-laboratory setting. This is hard to counteract, as it might be hard for the user to forget the fact that they are being supervised. One way to alleviate this might be to conduct remote tests where the users are in a familiar setting, like at home.

The facilitator may skew the feedback of the user, or cherrypick results. In the case of these user tests, the developer was also the test facilitator. This likely increases the chances for the facilitator biasing the results. The facilitator must take care to word themselves neutrally and not put words in the test subjects mouth. Even if the facilitator is consciously aware of these possible biases, they might unconsciously skew the results. The best course of action is, therefore, to outsource the user-testing to a person who holds no stake in the success of the system.

System Usability Scale

Users might have previous experience with similar tools, and therefore find the system being evaluated more usable than it is for relevant but inexperienced users. McLellan et al. wrote an article on the matter, finding that experienced users tended to provide more favourable ratings regardless of system domain [34]. To counteract this McLellan et al. suggest that practitioners of SUS tests ask users for their experience with systems in the same or similar domains. For this reason, questions on the profession and experience levels of the users were included in the form. They also suggest reporting the SUS scores from the users belonging to the target demographic. Positive bias was however, not an issue with more knowledgeable users in this test, as they were significantly more critical of the interface than casual users. This can be seen in the scores in table 5.6. Based on qualitative feedback from these users, the negative bias appears to stem from advanced users feeling that Listers user interface was overly simplified, and not providing enough features.

Several users reported that they had done the test using their mobile phones. This was not intended, as the interface has not been adequately tested and optimized for small screens. The usability scores of those who reported having used their phones do not significantly affect results. However, as the form never explicitly asked whether the user was on mobile or desktop, there are likely several unrecorded mobile users. This may have negatively affected the SUS scores of those users.

The first call to action was posted on the author’s personal Facebook. This message was seen by people identifying themselves as "friends" of the developer. This might bias the scores of the tool in a positive direction. These users were among the first to take the test, and most identify themselves as being unfamiliar with structured query languages. This might be the reason why this user group has rated the Lister tool higher than those having structured language experience. The effect of this bias should, however, be mitigated, as a large part of the test subjects come from outside the author’s network, through the WikiData community and ‘Kode24-klubben’ Facebook group.

5.3.4 Interpreting the Usability Test Scores

The SUS score of Lister, 74.45, places the interface at the 70th percentile of user interfaces in terms of usability. On this scale, anything above the 50th percentile is by definition an above-average usability score. Placing this score on Sauro et al.’s grading scale (table 2.2) gives Lister the usability grade ‘B’.

Chapter 6

Conclusion

6.1 Conclusion

Through a pre-study, it was found that contemporary semantic web user interfaces struggle with usability. Users of natural language interfaces often do not know what questions to ask of the system, or what its capabilities are. This difference between expectations of the NLI and the actual performance is called the *habitability problem*. Query builders, on the other hand, are explicit in their functionality to a fault. Overly complex forms overwhelm users, making even simple tasks a struggle.

In this paper, a hybrid approach is proposed, combining the best aspects of both NLI's and query builders. The vast majority of SPARQL queries are SELECT queries, and by designing an interface focusing on these can vastly reduce the complexity of the interface. Users can thus interact with and explore their data effortlessly, with low cognitive overhead. This reduced scope also solves the habitability problem, as natural language search bars are standard interfaces online. Users get explicit feedback, showing them that they can search for classes. By employing a *double-index* system, entity-, class-, and full-text queries can be mapped to relevant classes.

Before development started, an overview of the usability issues common in semantic web search engines was compiled, and an example solution was proposed. An iterative design and development process followed, involving users at two critical stages underway. This ensured that usability issues were found and corrected underway, and user feedback incorporated and validated.

This resulted in the *Lister* search engine. The web-application was tested using the System Usability Scale, which provides a good measure of how user-friendly the application is. The Lister SUS scores were significantly higher than the web-average, and the highest-rated interface among those surveyed in the thesis. Lister thus provides a user-friendly entryway to information retrieval on the semantic web and the trove of data contained therein.

6.2 Future Work

Purely natural language interfaces show great potential and may draw advantage of advances in machine learning to overcome the complexities of rule-based parsing systems. Better and more training data and standardized and high-quality benchmarks are crucial when it comes to building better machine-learning systems. Both training data and benchmarks are as of today not good enough to push the state of the art performance by much. Therefore, the development of training data and benchmarks should be of high priority. This field of development is of particular interest, as better structured language generation/translation might make not only SPARQL/SQL queries possible from natural language, but also programming languages.

In order to create the Lister search engine, a separate index was created to facilitate full-text search. Full-text search is a common feature of the 'regular' web and undoubtedly has many use cases on the semantic web. It is already implemented in frameworks like Apache Jena but is not yet part of the SPARQL standard. One method of implementing this could be to encode strings as part of a triple structure instead of as a literal. This would bring the performance of graph queries to full-text queries, enabling real-time search by strings.

Finally, the integration of list-retrieval features into traditional search engines is of particular interest. Traditional search engines are well established, and likely where users go to have their information needs fulfilled. Semantic searches are already implemented to some degree by leading search engines like Google, though mostly in the form of information cards about specific entities. By integrating a solution like *Lister* into existing search portals, the mainstream could benefit from the semantic web without changing their search habits. This would mean an information gain for the end-user and an excellent incentive for further work and improvements on existing triple-stores.

Bibliography

[1] E. Kaufmann and A. Bernstein, 'Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases,' *Journal of Web Semantics*, vol. 8, no. 4, pp. 377–393, 2010, Semantic Web Challenge 2009 User Interaction in Semantic Web research, ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2010.06.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826810000582.

[2] J. Klímek, P. Škoda and M. Nečaský, 'Survey of tools for linked data consumption,' *Semantic Web*, vol. 10, pp. 1–57, Aug. 2018. DOI: 10.3233/SW-180316.

[3] C. Bizer, T. Heath and T. Berners-Lee, 'Linked data - the story so far,' *Int. J. Semantic Web Inf. Syst.*, vol. 5, pp. 1–22, 2009.

[4] A.-S. Dadzie and M. Rowe, 'Approaches to visualising linked data: A survey,' *Semantic Web*, vol. 2, pp. 89–124, Jan. 2011. DOI: 10.3233/SW-2011-0037.

[5] T. D. Wilson, 'Information behaviour: An interdisciplinary perspective,' *Information processing & management*, vol. 33, no. 4, pp. 551–572, 1997.

[6] P. Pirolli and S. Card, 'Information foraging.,' *Psychological review*, vol. 106, no. 4, p. 643, 1999.

[7] G. Hachey and D. Ga, 'Semantic web user interfaces: A systematic mapping study and review,' 2012.

[8] A. Freitas, E. Curry, J. G. Oliveira and S. O'Riain, 'Querying heterogeneous datasets on the linked data web: Challenges, approaches, and trends,' *IEEE Internet Computing*, vol. 16, no. 1, pp. 24–33, Jan. 2012. DOI: 10.1109/MIC.2011.141.

[9] C. Bobed and E. Mena, 'Querygen: Semantic interpretation of keyword queries over heterogeneous information systems,' *Information Sciences*, vol. 329, pp. 412–433, Feb. 2016. DOI: 10.1016/j.ins.2015.09.013.

[10] E. Kaufmann, A. Bernstein and L. Fischer, 'Nlp-reduce: A "naıve" but domain-independent natural language interface for querying ontologies,' *4th European Semantic Web Conference (ESWC 2007)*, Jan. 2007.

[11] E. Kaufmann, A. Bernstein and R. Zumstein, 'Querix: A natural language interface to query ontologies based on clarification dialogs,' Jan. 2006.

[12] D. Damljanovic, M. Agatonovic and H. Cunningham, 'Freya: An interactive way of querying linked data using natural language,' in *The Semantic Web: ESWC 2011 Workshops*, R. García-Castro, D. Fensel and G. Antoniou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 125–138, ISBN: 978-3-642-25953-1.

[13] T. Soru, E. Marx, D. Moussallem, G. Publio, A. Valdestilhas, D. Esteves and C. Neto, 'Sparql as a foreign language,' Aug. 2017.

[14] A. Soylu, M. Giese, E. Jimenez-Ruiz, E. Kharlamov, D. Zheleznyakov and I. Horrocks, 'Optiquevqs: Towards an ontology-based visual query system for big data,' in *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, ser. MEDES '13, Luxembourg, Luxembourg: ACM, 2013, pp. 119–126, ISBN: 978-1-4503-2004-7. DOI: `10.1145/2536146.2536149`. [Online]. Available: `http://doi.acm.org/10.1145/2536146.2536149`.

[15] H. Vargas, C. Buil-Aranda, A. Hogan and C. López, 'Rdf explorer: A visual sparql query builder,' in *The Semantic Web – ISWC 2019*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois and F. Gandon, Eds., Cham: Springer International Publishing, 2019, pp. 647–663, ISBN: 978-3-030-30793-6.

[16] F. Haag, S. Lohmann, S. Siek and T. Ertl, 'Queryvowl: A visual query notation for linked data,' in *Revised Selected Papers of the ESWC 2015 Satellite Events on The Semantic Web: ESWC 2015 Satellite Events - Volume 9341*, New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 387–402, ISBN: 978-3-319-25638-2. DOI: `10.1007/978-3-319-25639-9_51`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-319-25639-9_51`.

[17] S. Heggestøyl, G. Vega-Gorgojo and M. Giese, 'Visual query formulation for linked open data: The norwegian entity registry case,' in *NIK*, 2014.

[18] O. Corby, R. Dieng and C. Faron-Zucker, 'Querying the semantic web with corese search engine,' in *ECAI*, 2004.

[19] M. Hildebrand, J. van Ossenbruggen and L. Hardman, '/facet: A browser for heterogeneous semantic web repositories,' in *The Semantic Web - ISWC 2006*, I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold and L. M. Aroyo, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 272–285, ISBN: 978-3-540-49055-5.

[20] M. Arenas, B. Cuenca Grau, E. Kharlamov, S. Marciuska, D. Zheleznyakov and E. Jimenez-Ruiz, 'Semfacet: Semantic faceted search over yago,' in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion, Seoul, Korea: ACM, 2014, pp. 123–126, ISBN: 978-1-4503-2745-9. DOI: `10.1145/2567948.2577011`. [Online]. Available: `http://doi.acm.org/10.1145/2567948.2577011`.

[21] S. Mazumdar, D. Petrelli, K. Elbedweihy, V. Lanfranchi and F. Ciravegna, 'Affective graphs: The visual appeal of linked data,' *Semantic web : interoperability, usability, applicability*, Aug. 2013. DOI: 10.3233/SW-140162.

[22] R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi and D. Petrelli, 'Hybrid search: Effectively combining keywords and semantic searches,' in *The Semantic Web: Research and Applications*, S. Bechhofer, M. Hauswirth, J. Hoffmann and M. Koubarakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 554–568, ISBN: 978-3-540-68234-9.

[23] G. Vega-gorgojo, M. Giese, S. Heggestøyl, A. Soylu and A. Waaler, 'Pepesearch: Semantic data for the masses,' *PLOS ONE*, vol. 11, e0151573, Mar. 2016. DOI: 10.1371/journal.pone.0151573.

[24] J. Brooke *et al.*, 'Sus-a quick and dirty usability scale,' *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[25] A. Bernstein, E. Kaufmann, C. Kaiser and C. Kiefer, 'Ginseng : A guided input natural language search engine for querying ontologies,' 2006.

[26] A. Bangor, P. T. Kortum and J. T. Miller, 'An empirical evaluation of the system usability scale,' *Intl. Journal of Human–Computer Interaction*, vol. 24, no. 6, pp. 574–594, 2008.

[27] J. Sauro and J. R. Lewis, *Quantifying the User Experience*, J. Sauro and J. R. Lewis, Eds. Boston: Morgan Kaufmann, 2012, ISBN: 978-0-12-384968-7. DOI: https://doi.org/10.1016/B978-0-12-384968-7.00004-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123849687000047.

[28] K. Elbedweihy, S. N. Wrigley and F. Ciravegna, 'Evaluating semantic search query approaches with expert and casual users,' in *The Semantic Web – ISWC 2012*, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein and E. Blomqvist, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 274–286, ISBN: 978-3-642-35173-0.

[29] V. Uren, Y. Lei, V. Lopez, H. Liu, E. Motta and M. Giordanino, 'The usability of semantic search tools: A review,' *Knowledge Eng. Review*, vol. 22, pp. 361–377, Dec. 2007. DOI: 10.1017/S0269888907001233.

[30] C. W. Thompson, P. Pazandak and H. R. Tennant, 'Talk to your semantic web,' *IEEE Internet Computing*, vol. 9, pp. 75–78, 2005.

[31] *Nlp progress*, http://nlpprogress.com/english/semantic_parsing.html, Accessed: 2019-11-14.

[32] A. Bonifati, W. Martens and T. Timm, 'An analytical study of large sparql query logs,' *The VLDB Journal*, pp. 1–25, 2019.

[33] J. Knapp, J. Zeratsky and B. Kowitz, *Sprint: how to solve big problems and test new ideas in just five days*. Simon & Schuster, 2016.

[34] S. McLellan, A. Muddimer and S. C. Peres, ‘The effect of experience on system usability scale ratings,’ *Journal of usability studies*, vol. 7, no. 2, pp. 56–67, 2012.

Appendix A

How To Set-Up Lister

Front-end

Requirements

- Node.js version ≥ 8.10
- npm version ≥ 5.6
- A host server (for production builds)

Procedure

The front-end has different configurations based on whether it is to be served in a development- or production-setting.

Development set-up

1. Clone the Lister front-end GitHub repository.
2. In the root directory of Lister, run the command `NPM install`.
3. Verify that the installation was successful.
4. In the root directory of Lister, run the command `NPM start`.
5. A browser window should open on the locally hosted front-end web application. If not, manually navigate to the url returned by the `NPM start` command.

Production set-up

1. Clone the Lister front-end GitHub repository (either locally or remotely on a web-host server).
2. In the root directory of Lister, run the command `NPM install`.
3. Verify that the installation was successful.
4. In the root directory of Lister, run the command `NPM run-script build`.
5. Move the contents of the `build` folder to the index of your server.
6. Verify that your web host is now serving the Lister front-end files. A *hard* refresh may be needed to avoid loading a cached version of the web-site.

Back-end

Requirements

It is recommended but not required to run the pre-processing notebooks using an Anaconda[1] Python distibution. Anaconda comes pre-packaged with almost all the python dependencies needed, and it makes setting up a virtual enviroment easy.

- Python version ≥ 3.6
- ElasticSearch version ≥ 7.7
- Node.JS version ≥ 8.10
- npm version ≥ 5.6

The following python packages are required, those that come with Anaconda are marked with an asterisk:

- tqdm*
- pandas* 1.0.3
- Dask* 2.16
- elasticsearch 7.7.0

Procedure

The back-end is comprised of two parts: The ElasticSearch indices and the Node backend. The set up for the ElasticSearch indices is as follows:

1. Download and unzip the latest WikiData JSON dump.
2. Download and install the required ElasticSearch version.
3. Set up and launch ElasticSearch
4. For pre-processing, follow the instructions in the `wikidata_preprocessing.ipynb` notebook.
5. To create and populate the ES indices, follow the instructions in the `populate_es.ipynb` notebook.

The set up for the Node.js backend is as follows:

1. Clone the Lister front-end GitHub repository.
2. In the root directory of Lister, run the command `NPM install`.
3. Verify that the installation was successful.
4. In the root directory of the Lister back-end, run the command `node backend.js`.
5. Verify that the set-up was successful by viewing the console logs, or by requesting `GET localhost:9200/` and `GET localhost:9200/cluster/health`.

[1]https://anaconda.org/

Appendix B

Master‘s Agreement

NTNU

Master`s Agreement

Faculty	**IE - Fakultet for informasjonsteknologi og elektroteknikk**
Institute	**Institutt for datateknologi og informatikk**
Programme code	**MTDT**
Course code	**194_TDT4900_1**

Personal information

Family name, first name	**Skaslien, Mats Jørgen**
Date of birth	**25.01.1994**
Email address	**matsjsk@stud.ntnu.no**

The Master`s thesis

Starting date	**15.01.2020**
Submission deadline	**10.06.2020**
Thesis working title	**Semantic Web Interface Project**
Thematic description	**Via en brukerorientert utviklingsprosess skal et grensesnitt til den Semantiske Webben skapes, så også folk uten relevant erfaring skal kunne ta Linked Data i bruk.**

Supervision and co-authors

Supervisor	**Trond Aalberg**
Any co-supervisors	**, - Merknad - Mulig medveileder Vedran Sabol ved TU Graz i Østerrike. Ikke bestemt per dags dato.**
Any co-authors	

Topics to be included in the Master`s Degree (if applicable)

Course code	Course name	Credits	Level	Term

Guidelines – Rights and Obligations

Purpose

Agreement on supervision of the Master's thesis is a cooperation agreement between the student, supervisor and the department that governs the relationship of supervision, scope, nature and responsibilities.

The master's program and the work of the master's thesis are regulated by the Act relating to universities and university colleges, NTNU's study regulations and current curriculum for the master's program.

Supervision

The student is responsible for

- Agre upon supervision within the framework of the agreement
- Set up a plan of progress for the work in cooperation with the supervisor, including the plan for when the guidance should take place
- Keep track of the number of hours spent with the supervisor
- Provide the supervisor with the necessary written material in a timely manner before the guidance
- Keep the institute and supervisor informed of any delays

The supervisor is responsible for

- Explain expectations of the guidance and how the guidance should take place
- Ensure that any necessary approvals are requested (REC, ethics, privacy)
- Provide advice on the formulation and demarcation of the topic and issue so that the work is feasible within the standard or agreed upon study time
- Discuss and evaluate hypotheses and methods
- Advice on professional literature, source material / data base / documentation and potential resource requirements
- Discuss the presentation (disposition, linguistic form, etc.)
- Discuss the results and the interpretation of them
- Stay informed about the progression of the student's work according to the agreed time and work plan, and follow up the student as needed
- Together with the student, keep an overview of the number of hours spent

The institute is responsible for

- Make sure that the agreement is entered into
- Find and appoint supervisor(-s)
- Enter into an agreement with another department / faculty / institution if there is a designated external supervisor
- In cooperation with the supervisor, keep an overview of the student's progress, an overview of the number of hours spent, and follow up if the student is delayed by appointment
- Appoint a new supervisor and arrange for a new agreement if
    - supervisor will be absent due to research term, illness, travel, etc., and if the student wishes
    - student or supervisor requests to terminate the agreement because one of the parties does not follow it
    - other circumstances make the parties find it appropriate with a new supervisor
- Notify the student when the guidance relationship expires.
- Inform supervisors about the responsibility for safeguarding ethical issues, privacy and guidance ethics
- Should the cooperation between student and supervisor become problematic for one of the parties, a student or supervisor may ask to be freed from the Master`s agreement. In such case, the institute must appoint a new supervisor

This Master`s agreement must be signed when the guidelines have been reviewed.

Approved by

Mats Jørgen Skaslien	Trond Aalberg	Berit Hellan
Student	**Supervisor**	**Institute**
14.01.2020	16.01.2020	07.02.2020
place and date	place and date	place and date

Mats Jørgen Skaslien

Lister: A Hybrid Approach for User-friendly Semantic Web Information Retrieval

NTNU
Kunnskap for en bedre verden