

Master thesis, Institute of Computer Science, Freie Universität Berlin

Human-Centered Computing (HCC)

Developing a Graphical User Interface for Generating Wikipedia Lists with Wikidata

Jakob Warkotsch

Supervisor: Prof. Dr. C. Müller-Birn

Secondary Supervisor: Prof. Dr. Lutz Prechelt

External Supervisor: Dipl.-Inf. Lydia Pintscher

Berlin, 05.01.2018

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 5. Januar 2018

Jakob Warkotsch

Abstract

Wikipedia is one of the most popular websites on the World Wide Web, providing free knowledge created by a community of volunteer editors. It is supported behind the scenes by Wikidata, the free and open knowledge base, another community-based Wikimedia project, which is a platform for the creation and management of structured data. Wikidata can be used to enrich Wikipedia articles with information and is especially useful outside of continuous text, for the display of tabular data or lists. At present, there are only programmatic solutions for the creation of lists using Wikidata's data. This thesis with the title "Developing a Graphical User Interface for Generating Wikipedia Lists with Wikidata" is about the development of a tool which aims to enable experienced Wikidata users to produce SPARQL queries for automated list generation without having to know SPARQL.

An extensive assessment of existing graphical SPARQL query builders was conducted, building upon an existing categorization to propose a hierarchical systematization of such tools based on their primary purpose and their user interface paradigm in order to learn from existing approaches and the possible trade-offs between simplicity and functionality. To find out which features of SPARQL and which graph patterns for queries needed to be supported by the tool, an analysis was performed on queries produced by the English and German Wikipedia community. Combining the insights from the analysis phase, a conceptual model was created with a novel mapping of Wikidata concepts to SPARQL constructs. A user-centered design approach was taken to iteratively test and improve the design with potential users with a low-fidelity and a high-fidelity prototype.

A functional version of the high-fidelity prototype was tested in a user study at the WikidataCon. Qualitative feedback was gathered using the thinking aloud method during the test sessions, and a standardized questionnaire was used to generate quantitative data, which also allowed the Query Builder's usability to be evaluated against the questionnaire's benchmark. Both the questionnaire results and the personal feedback at the conference suggested that the Wikidata Query Builder is going to be a useful and usable tool.

Zusammenfassung

Wikipedia, eine der meistbesuchten Webseiten im World Wide Web, ist eine Enzyklopädie aus freien Inhalten, die von freiwilligen Editoren gepflegt wird. Sie wird hinter den Kulissen von Wikidata, der freien Wissensdatenbank, unterstützt, die eine Plattform zur Erstellung und Verwaltung strukturierter Daten bietet. Wikidata kann benutzt werden um Wikipedia-Artikel mit strukturierten Inhalten anzureichern, was vor allem beim Erzeugen von Tabellen und Listen nützlich ist. Momentan existiert keine Lösung, die auch Benutzern ohne Programmierkenntnisse die automatische Generierung solcher von Wikidata erzeugten Listen erlaubt. In dieser Arbeit mit dem Titel "Developing a Graphical User Interface for Generating Wikipedia Lists with Wikidata" wird die Entwicklung einer Anwendung beschrieben, die es erfahrenen Wikidata-Benutzern erlaubt, SPARQL-Abfragen für die automatische Listenerstellung zu erzeugen ohne SPARQL-Kenntnisse vorauszusetzen.

Mit dem Ziel auf vorhandene Ansätze aufzubauen und deren Kompromisse zwischen Anwenderfreundlichkeit und Funktionsumfang zu erkennen, wurde eine ausführliche Untersuchung ähnlicher Programme zur Erstellung von SPARQL-Abfragen durchgeführt, und auf eine existierende Kategorisierung solcher Anwendungen aufgebaut, um eine Einordnung anhand des primären Anwendungsfalls und der Art der Benutzeroberfläche vornehmen zu können. Außerdem wurden Abfragen analysiert, die von Benutzern der Englischen und Deutschen Wikipedia erstellt wurden, um herauszufinden welche SPARQL-Funktionen und welche Graphstrukturen von der Anwendung unterstützt werden müssen. Aus diesen Untersuchungen wurden Anforderungen abgeleitet und ein neuartiges Design erstellt, das Wikidata-Konzepte auf SPARQL-Konstrukte abbildet. Mithilfe benutzerzentrierter Methoden wurden mehrere Prototypen erstellt, um das Design iterativ zu testen und zu verbessern.

Ein funktionierender Prototyp wurde auf der WikidataCon in einer Benutzerstudie getestet. Die Thinking-Aloud-Methode wurde während der Durchführung angewandt um qualitatives Feedback zu sammeln, und ein standardisierter Fragebogen wurde zur Erhebung quantitativer Daten genutzt, der eine Einschätzung der Benutzerfreundlichkeit anhand eines Vergleichsindex ermöglicht. Sowohl die Ergebnisse des Fragebogens, als auch die allgemeine Resonanz deuten darauf hin, dass mit dem Wikidata Query Builder eine nützliche und benutzerfreundliche Anwendung entwickelt wurde.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Research Goal	3
1.3	Research Approach	4
1.4	Research Context	6
1.5	Methodology	6
2	Theoretical Foundations	9
2.1	RDF	9
2.2	SPARQL	9
2.3	Wikidata Data Model	12
2.4	Wikidata RDF Representation	14
3	Related Work	17
3.1	Wikipedia, Wikidata, and Lists	17
3.2	Wikidata and Linked Data	18
3.3	SPARQL Analysis	19
3.4	Query Builder Tools	21
3.4.1	Overview	21
3.4.2	Categorization	23
3.5	Summary	23
4	Analysis	25
4.1	Listeria SPARQL Analysis	25
4.1.1	Data Collection	25
4.1.2	Analysis Setup	26
4.1.3	Data Cleaning	27
4.1.4	General SPARQL Features	27
4.1.5	Triple Pattern Structures	29
4.1.6	Common Property Paths	29
4.1.7	Commonly Used Wikidata Features	30
4.2	Requirements	31
4.2.1	Feature-specific Attributes	31
4.2.2	Interface-specific Attributes	32
4.2.3	Interaction-specific Attributes	32
4.3	State-of-the-Art Review	33
4.3.1	Categories	33
4.3.2	Exploratory Query Builders	34

4.3.3	Triple Pattern Based Query Builders	37
4.3.4	Attribute-based Assessment	39
4.4	Summary	43
5	Design	47
5.1	Conceptual Model	47
5.2	Low-Fidelity Prototype	51
5.3	High-Fidelity Prototype	53
5.4	Summary	58
6	Implementation	59
6.1	System Architecture	59
6.2	Technical Implementation	60
6.2.1	User Interface Components	60
6.2.2	State Management	62
6.2.3	SPARQL Query Generation	63
6.3	Summary	67
7	Evaluation	69
7.1	User Experience Questionnaire	69
7.2	Usability Test Setup	70
7.3	Results	72
7.3.1	Task Completion	73
7.3.2	UEQ Results	73
7.4	Summary	75
8	Discussion	77
8.1	Evaluation of this Work	77
8.2	Research Contribution	78
9	Conclusion	79
9.1	Summary	79
9.2	Limitations	80
9.3	Future Work	81
	Literature	85
	Appendix	93

1 Introduction

Wikidata¹ is a free and open knowledge base that anyone can edit [wikc]. It stores structured data about all kinds of things in a way that is human-readable, and also accessible through application programming interfaces. Like Wikipedia², Wikidata is a collaborative project, and all its content is created and managed by volunteer editors.

Wikidata is not only useful within the scope of Wikimedia projects. Most notably, it became part of Google’s knowledge graph after the shutdown of the Freebase knowledge base³ [fre], and part of the data from Freebase has been migrated to Wikidata [PTVS⁺16].

The idea for Wikidata emerged in the process of finding a solution for the increasing amount of data that was duplicated and needed to be maintained and kept consistent across multiple languages in the text-based Wikipedia platform. It became evident that a central structured data management solution was needed, and so the Wikidata project was started as part of the Wikimedia movement⁴. Both Wikipedia and Wikidata structure their content as web pages – Wikipedia has article pages and Wikidata has entity pages [VK14]. Wikidata is a multilingual knowledge base and each entity page contains information represented in multiple languages, whereas a Wikipedia article page is always written in only one language.

Wherever an article on Wikipedia has a corresponding Wikidata item, the data from this item or related items can be used within the article. A common place for finding Wikidata used on Wikipedia are the so called *infoboxes* that can be found on the right upper corner of many Wikipedia articles, where a collection of fundamental data is presented to the user in tabular form.

Just like infoboxes, lists on Wikipedia rarely contain continuous text, and they too could be automatically generated based on data that is already available on Wikidata. Wikipedia lists powered by Wikidata are a powerful idea, that could greatly reduce the amount of maintenance work for many Wikipedia editors and improve the content on both platforms. Especially smaller Wikipedias with few editors can greatly benefit from such a solution – if lists can easily be reused for different languages, then a small number of editors can create more and higher quality content.

One missing part to make this become reality is a tool which enables people, who are familiar with Wikidata but who do not know the semantic query language SPARQL, to create such list. This thesis is about the design of a

¹https://www.wikidata.org/wiki/Wikidata:Main_Page

²<https://www.wikipedia.org/>

³<https://en.wikipedia.org/wiki/Freebase>

⁴<https://www.wikimedia.org/>

1.1. Motivation

graphical user interface and the development of a working prototype of the Wikidata Query Builder for list generation.

1.1 Motivation

Wikipedia is one of the most well known and most frequently visited websites in the world [wikg]. It is the largest encyclopedia in existence and the go-to resource for many people wanting to learn more about a certain topic.

Aside from articles about specific topics, such as the article about Mount Fuji⁵, there are also list articles about related topics such as the list of volcanoes in Japan⁶. What is special about these articles, is that their focal point is a list or table which contains no continuous text, but language independent data – in the case of the list of Japanese volcanoes it contains the volcano’s name, its elevation above sea level, coordinates and date of its last eruption.

In many cases all the data that is part of Wikipedia list articles is already present on Wikidata and a lot of the editors’ time and work could be saved if Wikidata was used to generate such lists instead of having them manually created, maintained, and duplicated across multiple Wikipedias in different languages. Ideally, with all the data for lists centrally maintained in Wikidata, there would be no need to create a list more than once. A change of a language parameter for labels of the concepts in the lists would be necessary to include the same list in a Wikipedia for a different language. The lists could also update automatically once the central data on Wikidata changes, thereby reducing the maintenance work and keeping all lists consistent in all places. Finally, by using more of Wikidata’s data on Wikipedia, the symbiosis between the two projects is strengthened as one edit to improve such a list improves the content of Wikipedia and Wikidata at the same time.

Two steps are left to make Wikidata-generated lists on Wikipedia a reality:

1. Enabling support for Wikidata lists on Wikipedia

This problem is out of scope thesis. Aside from the need for a technical solution, there is an ongoing discussion with the Wikipedia community about how such lists can be integrated into articles without causing confusion. Questions that have to be tackled include how changes from Wikidata could be reflected in the edit history of the article, and how manual additions and edits to an automated list can be handled.

Even though automatically generated lists on Wikipedia are not an officially supported feature yet, the community has come up with ways to experiment with it. Using a custom Wikitext template⁷, queries written in the semantic query language SPARQL can be included in wiki

⁵https://en.wikipedia.org/wiki/Mount_Fuji

⁶https://en.wikipedia.org/wiki/List_of_volcanoes_in_Japan

⁷https://en.wikipedia.org/wiki/Template:Wikidata_list

pages which get periodically crawled by the Listeria Bot⁸. Wikipedia bots are programs that have been granted edit rights on Wikipedia and are used to facilitate repetitive and mundane tasks [wikf]. This particular bot parses the query, sends the request to Wikidata’s SPARQL endpoint and updates the list on the page accordingly. Despite being a somewhat make-shift solution and the ongoing discussion [lisa] about its problems, this bot has already been proven to be a powerful tool. It is used by the community for facilitating maintenance work and within the popular Wiki Loves Monuments⁹ contest to generate lists of monuments within certain areas.

2. Providing a way for Wikidata users to generate lists

This is what this thesis is mainly about. As of now, users wanting to automatically generate such lists are limited to tools like the Wikidata Query Service¹⁰, the Wikidata Toolkit¹¹ and the Wikidata API¹² which all require programming skills. Especially the Query Service has gained in popularity with millions of SPARQL queries executed every day [wdq] and providing various result visualizations out of the box.

With the goal of enabling more people to create their own Wikidata queries, a Wikidata Query Builder, which functions as a simplified graphical interface to the Query Service and outputs SPARQL queries, will be developed. Building the Query Builder on top of the Query Service provides two significant advantages:

1. The result view of the Query Service can be used for immediate feedback
2. The resulting SPARQL query can be used to generate lists on Wikipedia using the experimental Listeria Bot

Providing the community with a functional prototype for list generation has the potential to get more people, especially those without SPARQL knowledge, involved in the discussion around list generation and also get more concrete feedback on the users’ needs and use cases for such a tool.

1.2 Research Goal

The main goal of this work is to develop a list generation tool for expert Wikidata users, who are not SPARQL experts. In order to ensure that this

⁸<https://en.wikipedia.org/wiki/User:ListeriaBot>

⁹<https://www.wikilovesmonuments.org/contest/>

¹⁰<https://query.wikidata.org/>

¹¹https://www.mediawiki.org/wiki/Wikidata_Toolkit

¹²<https://www.wikidata.org/w/api.php>

1.3. Research Approach

tool is both useful and usable, a considerable amount of user research and analyses of existing query builder tools were done.

The problem with tools, as Norman et al.[ND86] put it, is often that they are either too simple and require too much skill from the user, or they are too intelligent and leave the user with a feeling of lack of control. The goal for the Query Builder prototype is to find a middle ground in this regard – a tool that is simple and understandable, yet designed in a way that the user does not have to think about low level constructs of SPARQL and Wikidata’s RDF schema that diverge from the concepts Wikidata users are familiar with.

With this in mind, a secondary goal was a thorough analysis of existing SPARQL query building applications to analyze and categorize the different interface paradigms and use cases of state-of-the-art graphical interfaces for building queries on linked datasets. This analysis yielded an overview of different approaches and their respective strengths and weaknesses.

1.3 Research Approach

The following section describes the development and design process of the Query Builder. Each of the steps depicted in figure 1.1 yielded results that were required for the next step. The loop back from the evaluation step to the design step visualizes how changes to the user interface were done iteratively – evaluating each change with users and adapting based on the given feedback.

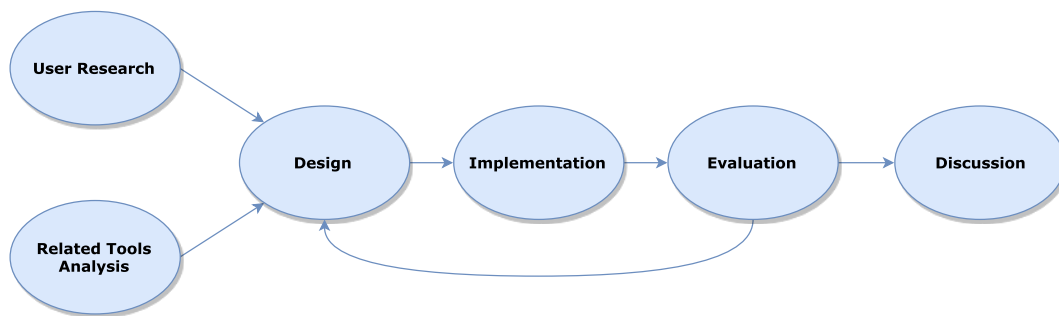


Figure 1.1: Research approach structure

Related Tools Analysis

With the goal of building upon existing approaches and avoiding to reinvent the wheel, related query builders were researched and categorized. This provided an overview of possible options for user interface approaches.

User Research

In order to understand what features an application needs to have to be useful, one must first understand what its target audience intends to do with it. Since the goal is to produce SPARQL queries that generate lists for Wikipedia, the existing SPARQL queries from the Listeria Bot templates on Wikipedia were an obvious choice for a data source.

These queries were analyzed with three questions in mind:

1. What is the minimal set of SPARQL features that covers most queries?
2. To what extent does each of the features need to be supported?
3. What Wikidata specific features are used often and can be simplified?

Additionally, based on the outcome of the analysis of the SPARQL use and the related tools, a set of desirable feature, interface and interaction attributes was defined.

Design

By combining the results from the previous two steps, it was possible to see which of the approaches from related projects are compatible with the user needs that were revealed in the analysis phase.

After learning what the simplest possible interface is that still supports most of the SPARQL queries for lists that were analyzed, low fidelity prototypes were developed for testing out possible variations of the design that has been decided upon. Several informal tests were conducted to eliminate major usability issues up front and iteratively improve the design of the user interface.

Implementation

With the basic structure of the user interface set, the next step was to begin with the implementation of the high fidelity prototype. This was not intended to be a polished product, but rather a working proof of concept with all the major features in place in order to gather good qualitative feedback from real users.

Evaluation

At the WikidataCon¹³, the first ever Wikidata conference, the high fidelity prototype was tested with 18 individuals. Each test session consisted of three scenarios of increasing difficulty that would let the potential future users explore the features of the user interface. Qualitative feedback was gathered by using the thinking aloud testing method during the scenarios and afterwards

¹³https://www.wikidata.org/wiki/Wikidata:WikidataCon_2017

1.5. Methodology

each test subject was asked to fill out the User Experience Questionnaire¹⁴ for quantitative feedback.

The feedback from users at the WikidataCon has shed light on some design flaws that went unnoticed during the initial low fidelity prototype testing. Three elements of the user interface that caused issues during the test session had to undergo another feedback round with shorter and more informal test sessions shortly after the conference.

Discussion

The final step is there to discuss whether the defined goals were achieved and reflect on possible shortcomings and limitations in this work.

1.4 Research Context

This work is an external thesis that is supervised by the Wikidata software development team at Wikimedia Deutschland¹⁵. The main objective of the Wikidata team, the largest of the three software development teams at Wikimedia Deutschland, is the maintenance and extension of Wikibase¹⁶, the open source software that powers Wikidata.

The Wikidata project was started in 2012 as a collaboratively edited database of the world's knowledge in order to centrally store the growing amount of structured data used within the Wikipedia project [wike]. Since then, and especially after the release of its SPARQL endpoint [spa], Wikidata has become a notable participant in the linked open data web.

Similar reputable structured knowledge projects include DBpedia, Yago, and Freebase. DBpedia is a project that is based upon structured knowledge that is extracted from Wikipedia [ABK⁺07]. Compared to Wikidata, the focus is more on creating a structured knowledge graph from the extracted data, and less community-centric than Wikidata. Freebase is a knowledge base developed by Google with the goal of providing data for Google's knowledge graph¹⁷. It has since been shut down in favor of Wikidata [PTVS⁺16]. Like DBpedia, Yago is a database that extracts data from Wikipedia and other sources, and claims to have high standards for coverage and quality [SKW07].

1.5 Methodology

In order to achieve the goal of this thesis of creating a highly usable user interface for generating lists for Wikipedia by building SPARQL queries for

¹⁴<http://www.ueq-online.org/>

¹⁵<https://wikimedia.de/wiki/Hauptseite>

¹⁶<http://wikiba.se/>

¹⁷<https://www.google.com/intl/bn/insidesearch/features/search/knowledge.html>

Wikidata, methods and best practices from the user-centered design methodology will be applied. By involving users in the design process, it is ensured that a usable user interface will be designed and continuously improved by following an iterative process [AMKP04]. The typical stages of analyzing, designing, implementing, and evaluating are reflected in the research approach of this thesis.

The *Formative Evaluation* [LFH17] research method will be applied during the design and prototyping phase. Iterative user testing of low-fidelity and high-fidelity prototypes with test subjects from the Wikipedia and Wikidata community will ensure that a usable and useful application is built. During this process, the established thinking aloud testing method [Hol05] will be used for gathering qualitative feedback.

A *Summative Evaluation* follows after the initial implementation of the working high-fidelity prototype is completed. The *User Experience Questionnaire* (UEQ) [LHS08] was used to let users assess the Query Builder's perceived user experience, which in the case of the UEQ, is divided into the attributes attractiveness, perspicuity, efficiency, dependability, stimulation and novelty. The more detailed UEQ with its 26 questionnaire items was chosen instead of the *Standard Usability Scale* (SUS) [B⁺96] since SUS was considered too generic. Having more a more detailed assessment was preferable in order to get feedback that is easier to interpret.

1.5. Methodology

2 Theoretical Foundations

Aside from thoroughly analyzing the user needs and related projects, a solid understanding of the underlying technologies and data representations of Wikidata are important to build a specialized SPARQL query building interface. This chapter first describes the basics of the linked data technologies RDF and SPARQL, and then outlines how data is modeled in Wikidata and how its data model is represented in RDF.

2.1 RDF

The *Resource Description Framework* (RDF) [CK04] was created to represent information on the web. It describes data as a set of triples of subjects, predicates and objects which form a graph with subjects and objects as nodes and predicates as edges, directed from subject to object, as shown in figure 2.1. Triples represent a statement of the relationship, described by the predicate, between the subject and the object.

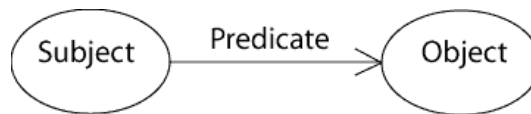


Figure 2.1: RDF triple, image adapted from [CK04]

URIs are used to identify subjects, predicates and objects in RDF. As opposed to URLs, that are addresses for documents on the web, URIs are meant to provide a more generic way to identify entities [BHBL09]. Following the Linked Data best practices, these URIs should provide further information about the underlying concept when looked up.

Nodes that are not represented by a URI are either literals or blank nodes. Literals are used for referencing simple values in RDF such as numbers, dates or strings. Blank nodes are resources that are unidentified and mostly serve structural purposes.

2.2 SPARQL

While RDF is the go-to data format for representing information on the semantic web, SPARQL is arguably the most popular choice for querying RDF [PS⁺06]. Officially released in 2008, SPARQL is a relatively new query language with a syntax that outwardly resembles that of SQL.

2.2. SPARQL

SPARQL queries start with prefix definitions that allow for URIs within the query to be abbreviated to the defined prefix e.g. with `PREFIX ex: <http://example.org/>` we can use `ex:Author` instead of `<http://example.org/Author>`. The prefix definitions are followed by one of the four possible query forms `SELECT`, `CONSTRUCT`, `ASK` or `DESCRIBE`. `SELECT`, the query form that returns the variables that were bound in a query pattern match, is the one relevant for this work.

The main part of the query, the `WHERE` clause, provides a graph pattern that is matched against the data graph. Given a data source with nodes representing books that have an author predicate with the URI `http://example.org/Author` and nodes for their authors, `http://example.org/DouglasAdams` being one of them, then the query in listing 2.1 returns a list of URIs of books by Douglas Adams.

```
1 PREFIX ex: <http://example.org/> .
2
3 SELECT ?book
4 WHERE {
5     ?book ex:author ex:DouglasAdams .
6 }
```

Listing 2.1: A simple SPARQL query

The triples in the graph pattern of the query are like RDF triples where subject, predicate and object may also be a variable. The query returns a result if there exists a subgraph in the data graph that is equivalent to the query graph pattern when variables are substituted for RDF terms. The RDF terms filled in for the variables in the graph pattern are part of the result set if the variable was part of the `SELECT` clause.

Query graph patterns can be arbitrarily complex. A more complex query can be seen in listing 2.2, visualized in figure 2.2, which results in a list of books that are set in the author's place of birth. Since all three nodes' variables are specified in the `SELECT` clause, the result set will contain the URIs of the books, the authors and the places.

```
1 PREFIX ex: <http://example.org/> .
2
3 SELECT ?book ?author ?place
4 WHERE {
5     ?book ex:author ?author .
6     ?author ex:placeOfBirth ?place .
7     ?book ex:setting ?place .
8 }
```

Listing 2.2: Books set in the author's place of birth

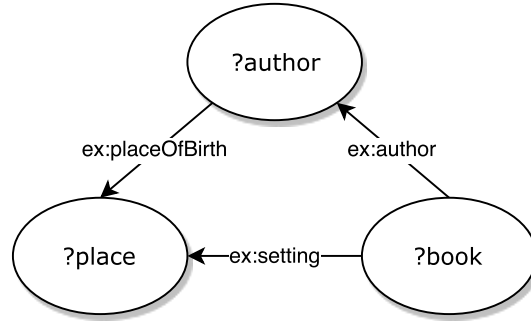


Figure 2.2: Same query from listing 2.2, visualized

In related literature [HHR⁺16, SAH⁺15], query graphs are analyzed based on concepts from network analysis, or by their types of variable joins. The commonly analyzed patterns are *sink*, *path*, *star* (or *source*), and *cycle*. Related to the graph patterns, variable joins are classified into *subject-subject* joins, *subject-object* joins, and *object-object* joins.

Stars are nodes with only outgoing edges. They are formed by SPARQL queries with *subject-subject* joins, as seen in figure 2.2 in the `?book` variable, and lines 5 and 7 of listing 2.2.

Paths are formed by nodes with at least one ingoing and at least one outgoing edge, for example the `?author` node in figure 2.2. Similarly, *subject-object* joins are those where the subject variable of one triple pattern is identical to the object variable of another triple pattern.

Sinks are nodes with only ingoing edges. Sinks with multiple ingoing edges are built with triple patterns with object-object joins. One example of a sink is the `?place` variable in figure 2.2, built from the object-object join on lines 6 and 7 in listing 2.2.

Cycles are patterns where any node can reach itself by following the directed predicate edges. Figure 2.2 does not contain a cycle.

Graph patterns can be grouped with `OPTIONAL`, `UNION` to express optional or alternative parts of a query. Subgraphs matched by patterns grouped with `MINUS` are removed from the result set.

In order to express queries like "Books with titles containing 'Harry Potter'" or "Mountains that are higher than 5000m", the `FILTER` feature can be used to test such conditions. The second example could be done as shown in listing 2.2.

```

1 PREFIX ex: <http://example.org/> .
2
3 SELECT ?mountain
4 WHERE {
5     ?mountain ex:elevationAboveSeaLevel ?elevation .
6     FILTER(?elevation > 4000)
7 }
  
```

2.3. Wikidata Data Model

SPARQL also supports query modifiers like `ORDER BY`, `LIMIT`, `OFFSET` and `DISTINCT`, as well as aggregates such as `COUNT` in combination with `GROUP BY`. These features work similar to their counterparts in relational query languages.

One feature of SPARQL that, as chapter 4 shows, is used more often by the Wikidata community than in most other projects, are *property paths*. Property paths are syntactically similar to regular expressions and allow the expression of a path through which a subject is connected to an object. One commonly seen property path in queries on the Wikidata Query Service is `wdt:P31/wdt:P279*` which means "instance of any subclass of". Subjects connected to objects through this property path are either direct instances of the object or instances of subclasses of subclasses... of the object. One example for this would be "instances of subclass of cat" (`?cat wdt:P31 / wdt:P279* wd:Q1461`) where one would possibly want to find direct instances of "cat" as well as instances of "British Shorthair"², which is modeled as a subclass of cat in Wikidata.

2.3 Wikidata Data Model

The following is a brief overview of relevant parts of Wikidata's data model. These concepts are fundamental to the design of the Query Builder, since one of the main ideas is to present the user with familiar terms from Wikidata and map them to SPARQL code behind the scenes.

Items

Wikidata models all concepts, topics and objects as *items* [hela], each of which is identified by an item identifier, which consists of the letter "Q" followed by a number. Q42³ for example, as seen in figure 2.3, identifies the item about the English writer and humorist, Douglas Adams.

Items can have a *label* and a *description* per language, and multiple *aliases*. *Sitelinks* are used to connect an item to other Wikimedia projects, e.g. connecting the item of Barack Obama⁴ with the article about him on the English Wikipedia⁵.

Properties

A *property* combined with a value can be used to describe items with statements. Like items, properties have their own pages and unique numeric IDs within Wikidata, prefixed with a "P". They also have labels, descriptions and aliases.

¹<https://www.wikidata.org/wiki/Q146>

²<https://www.wikidata.org/wiki/Q29273>

³<https://www.wikidata.org/wiki/Q42>

⁴<https://www.wikidata.org/wiki/Q76>

⁵https://en.wikipedia.org/wiki/Barack_Obama

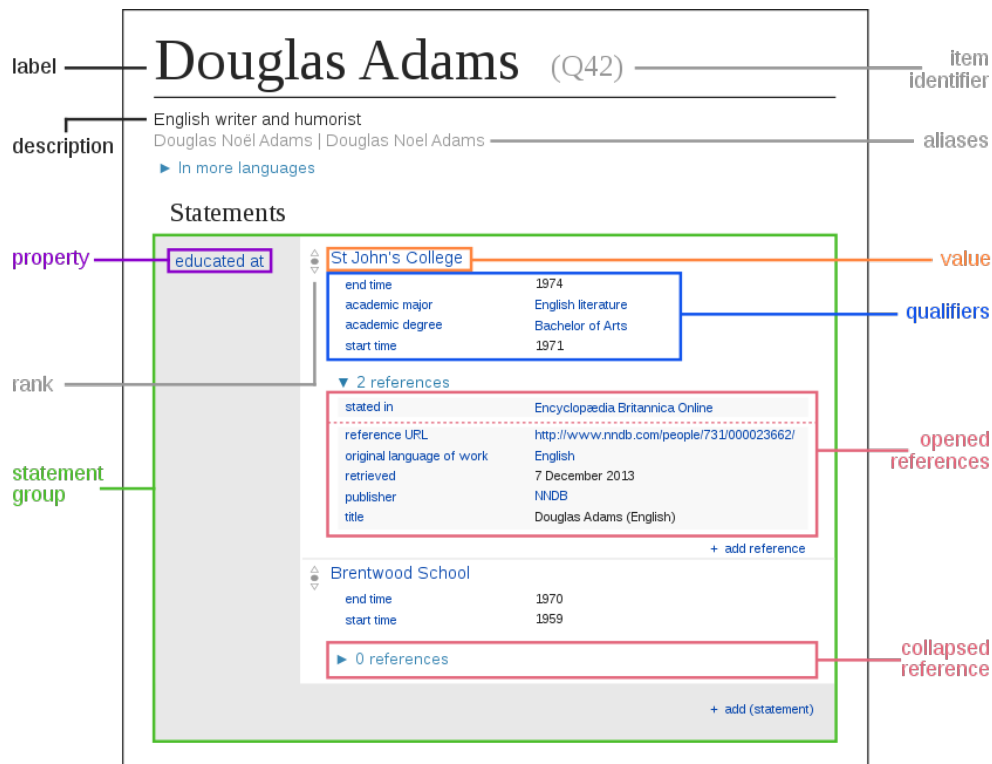


Figure 2.3: Wikidata item about Douglas Adams (CC0 by Kritschmar)

An important part of a property is their datatype, which defines the type of values that statements with this property can have. The "date of birth"⁶ property for example has the datatype "point in time" which limits statement values to date and time values, whereas the "head of government"⁷ has the datatype "item" to link e.g. a city item to the item of its mayor.

Statements

Statements are used to describe the information that is known about an item[hele]. Through statements Wikidata items can be linked together and have their relationship described by the property of the statement.

Table 2.3 gives an example of of the item of Berlin and the item about Michael Müller and how they are linked through a "head of government" statement.

⁶<https://www.wikidata.org/wiki/Property:P569>

⁷<https://www.wikidata.org/wiki/Property:P6>

2.4. Wikidata RDF Representation

item	property	value
Berlin	instance of	city
Berlin	head of government	Michael Müller
Michael Müller	date of birth	9 December 1964
Michael Müller	occupation	politician

Qualifiers

Qualifiers are used to expand on, annotate, or add context to a statement [helb]. Similar to statements, the main part of a qualifier is a property-value-pair, but instead of describing an item, it describes a statement.

The statement claiming that the head of government of the United States of America is Barack Obama is only true with the added qualifiers for start time (20 January 2009) and end time (20 January 2017).

References

Like qualifiers, *references* can optionally be added to statements. They specify the source that backs up the statement they belong to[held].

Ranks

Ranks provide a way to indicate whether the value of the statement they belong to is *preferred*, *normal*, or *deprecated* [helc]. The usefulness of this feature becomes evident when considering historical information, where the latest value for a certain property’s statement is typically the preferred one – in the case of a population measurement for a country for example. There is value in keeping track of outdated values for the population, to observe its growth for example, but the latest measurement is typically the one of highest interest.

2.4 Wikidata RDF Representation

As wikidata.org itself does not use a database that supports RDF or SPARQL, the data for the SPARQL endpoint that is part of the Wikidata Query Service has to be exported to RDF and mapped from its internal representation to the RDF format. The mapping is described by Malyshev in [rdf], which in turn is largely based on the format introduced by Krötzsch, Vrandečić et al. in [EGK⁺14]. The nodes that represent items and their prefixes are visualized in figure 2.4, the edges, labeled with their prefixes, depict the predicates between nodes. The examples in this section use the turtle⁸ format.

Simple item-property-value statements are represented as `wd:[item ID]` `wdt:[property ID]` `[value]` in the RDF format. So, "Angela Merkel is

⁸<https://www.w3.org/TR/turtle/>

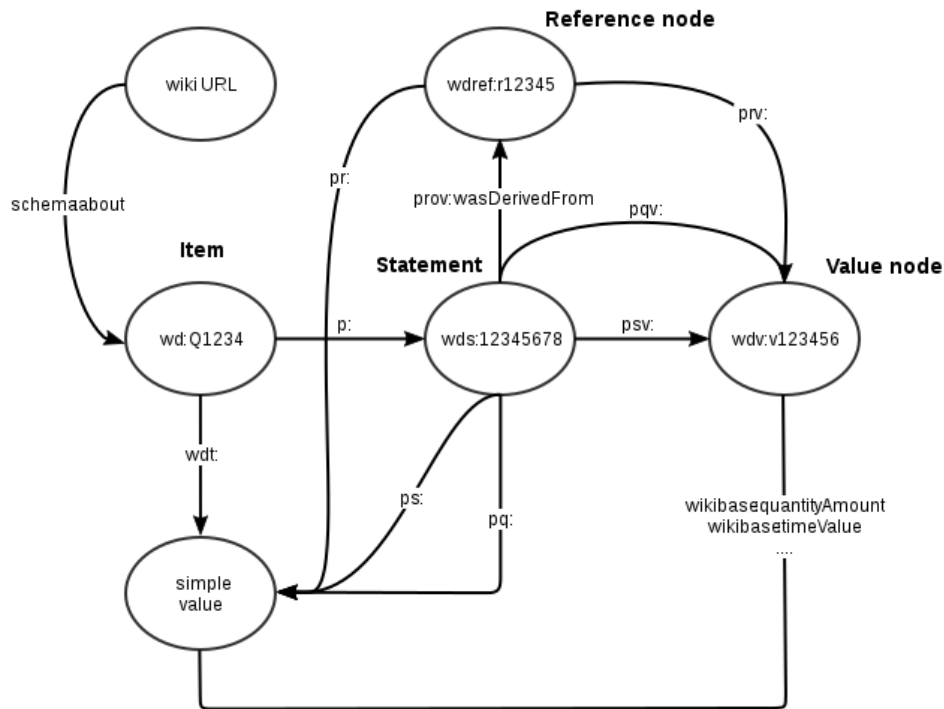


Figure 2.4: Wikidata item data representation in RDF (CC0 by Smalyshev)

the head of government of Germany” is represented as `wd:Q5679 wdt:P610 wd:Q18311`.

In order to describe an item’s statement with its qualifiers and additional information, statements needed to be represented as their own nodes. The triple `wd:Q3 p:P2 wds:Q3-4cc1f2d1-490e-c9c7-4560-46c3cce05bb7` represents the fact that the item Q3 has a statement for the property P2. Every statement node is connected to its value node via a `ps:[property ID]` predicate, and to its qualifier value nodes via `pq:[property ID]`.

The following example shows how it is represented that the item about Germany has a statement for ”head of government” (P6) with the statement’s value being the item about Angela Merkel, and a statement qualifier with the ”start date” property (P580) that has the value ”Nov 22, 2005”.

```
1 wd:Q183 p:P6 wds:Q183-d0db3461-4291-0b36-4092-c40d14699212 .
2
3 wds:Q183-d0db3461-4291-0b36-4092-c40d14699212 ps:P6 wd:Q567 ;
4 pq:P580 "2005-11-22T00:00:00Z"^^xsd:dateTime .
```

⁹<https://www.wikidata.org/wiki/Q567>

¹⁰<https://www.wikidata.org/wiki/Property:P6>

¹¹<https://www.wikidata.org/wiki/Q183>

2.4. Wikidata RDF Representation

Sitelinks are directly connected to items through the `schema:about` predicate, e.g. `<https://en.wikipedia.org/wiki/Duck> schema:about wd:Q3736439`.

3 Related Work

This chapter gives an overview of related work. A short introduction to the context of this thesis, Wikipedia, Wikidata, and the semantic web, is presented in the first section, followed by an outline of studies that are relevant to the research approach. Finally, a review of related SPARQL query building tools, their goals and approaches is provided in the last section.

3.1 Wikipedia, Wikidata, and Lists

As mentioned, Wikipedia already greatly benefits from the multilingual data that is stored in Wikidata. It finds use within the continuous text of articles, and even more commonly inside the infoboxes that provide general data about the article's subject.

Aside from that, other notable work has been done by Kaffee [kaf] for the Article Placeholder extension¹ which shows an article placeholder with data from Wikidata in the case that a user looked up a topic for which there does not exist an article on the accessed Wikipedia. This is especially useful for Wikipedias with a smaller contributor base where the chances are higher that many common topics are covered by Wikidata items but not on the Wikipedia in question.

Just like infoboxes or data-driven article placeholders, lists on Wikipedia articles are not dependent on continuous text and could in many cases be generated from Wikidata. In fact, automatic list generation has been part of the agenda since Wikidata's first announcement in March 2012 [wike], yet it is still not supported. Since then, Manske [lisb] has created the Listeria Bot with the ability to turn SPARQL queries into lists on MediaWiki articles.

This bot uses the Wikidata list template² which accepts a SPARQL query, result columns and additional options as parameters, and fills the space between the "Wikidata list"-template and "Wikidata list end"-template with an automatically generated list. Examples of the experimental use of the Listeria template can be seen on some Wikipedia user pages as seen in figure 3.1.

```
1 {{Wikidata list
2   |sparql=SELECT ?item WHERE { ?item wdt:P31 wd:Q3305213 . ?item wdt:
      P195 wd:Q214867 }
3   |columns=P18,label:Article,P170,P571,P217,P180,P127
4   |sort=571
5   |thumb=128
```

¹<https://www.mediawiki.org/wiki/Extension:ArticlePlaceholder>

²https://en.wikipedia.org/wiki/Template:Wikidata_list

3.2. Wikidata and Linked Data

```
6 }}
7 -- this part will get replaced by the Listeria bot --
8 {{Wikidata list end}}
```

Listing 3.1: Wikidata list template example for generating a list of paintings from User:Jane023³

This example from listing 3.1 shows how the template is used to automatically generate a list of paintings from the National Gallery of Art. The `sparql` parameter is used for the SPARQL query, `columns` specifies result columns of the list, a `sort` parameter specifies the order and `thumb` defines the width of thumbnails of the images.

User:Jane023/Paintings in the National Gallery of Art

From Wikipedia, the free encyclopedia
< User:Jane023

This list is automatically generated from data in Wikidata and is periodically updated by Listeria bot. Automatically update the list now | SPARQL | Find images
Edits made within the list area will be removed on the next update!

painting [edit source]

image	Article	creator	inception	inventory number	depicts	owned by
	<i>Madonna and Child with Saint John the Baptist, Saint Peter, and Two Angels</i>	anonymous	1280	1952.5.60	Child Jesus boy Mary woman Peter John the Baptist	
	<i>The Nativity with the Prophets Isaiah and Ezekiel</i>	Duccio di Buoninsegna	1308	1937.1.8	Child Jesus boy Mary woman	
	<i>The Calling of the Apostles Peter and Andrew</i>	Duccio di Buoninsegna	1308	1939.1.141	Peter Andrew Jesus Christ	
	<i>The Coronation of the Virgin</i>	Master of the Washington Coronation	1324	1952.5.87		
	<i>Madonna with the Child</i>	Giotto di Bondone	1325	1939.1.256	Mary woman Child Jesus boy mother	

Figure 3.1: Example of the list produced by the template in listing 3.1

3.2 Wikidata and Linked Data

Berners-Lee [BL06] describes 4 best practices for platforms on the web of data:

1. URIs for naming things
2. HTTP URIs for looking up names

3. When looking up a URI, data should be presented using semantic web technologies like RDF or SPARQL
4. Discoverability should be promoted by providing URIs to other things

The Wikidata project followed those practices from the beginning with unique identifiers for entities, that can be looked up via HTTP and included additional information by linking related concepts' URIs [VK14]. The missing link came with the release of the SPARQL endpoint at the end of 2015 [spa] with the RDF schema based on the work from Krötsch et al. [EGK⁺14] that finally enabled people to query Wikidata's data graph with SPARQL.

3.3 SPARQL Analysis

Analyzing the SPARQL queries from the Listeria Bot templates has been a major part of the user research and shaped the design of the structural user interface design of the Query Builder.

Hernández et al. [HHR⁺16] have done an analysis of the SPARQL features as well as graph patterns used in the 94 example queries on query.wikidata.org in their paper about comparing different databases for querying Wikidata. Their analysis of SPARQL features use is summarized in table 3.1. In addition to the numbers given in the table, they claim that **FILTER** was used often and sub-queries used occasionally.

Feature	out of 94 total	in %
ORDER BY	45	48
OPTIONAL	38	40
LIMIT	21	22
UNION	10	11
GROUP BY	27	29
BIND	10	11
NOT EXIST	5	5
Property Path	18	19

Table 3.1: Analysis of Wikidata Query Service example queries, adapted from Hernández et al. [HHR⁺16]

Analyzing the basic graph patterns within the queries, they found that most query patterns (80 out of 94) form trees and only 3 queries from the examples used variables in the predicate position. Unfortunately, not much information was given on the use of Wikidata-specific structures except that property paths for handling subclass hierarchies are frequently used and that 11 out of 94 queries involved statement qualifiers.

A larger dataset, not related to Wikidata, that has been reviewed extensively is the Linked SPARQL Queries Dataset (LSQ) [SAH⁺15]. The SPARQL

3.3. SPARQL Analysis

Feature	DBpedia	LGD	SWDF	BM	Overall
UNION	4.42%	9.65%	32.71%	0.00%	7.64%
OPTIONAL	36.20%	25.10%	25.32%	0.00%	31.78%
DISTINCT	18.44%	22.25%	45.40%	100.00%	23.30%
FILTER	23.47%	31.10%	0.95%	0.00%	23.19%
BIND	0.00%	0.00%	0.01%	0.00%	< 0.01%
(NOT) EXIST	< 0.01%	0.00%	< 0.01%	0.00%	< 0.01 %
REGEX	2.90%	1.25%	0.06%	0.00%	2.22%
LIMIT, OFFSET, ORDER BY	1.04%	60.44%	33.27%	0.00%	18.12%
Sub-Query	0.00%	0.01%	0.02%	0.00%	0.01%

Table 3.2: LSQ SPARQL feature analysis, adapted from Saleem et al. [SAH⁺15]

queries have been extracted from the logs of four public SPARQL endpoints: DBpedia, Linked Geo Data (LGD), Semantic Web Dog Food (SWDF) and the British Museum (BM). Unlike the data from Hernández [HHR⁺16], the queries from LSQ contain not only **SELECT** (91.6%), but also the other query forms **CONSTRUCT** (1.2%), **DESCRIBE** (2.3%), and **ASK** (4.9%).

Saleem et al. [SAH⁺15] also assessed the usage of SPARQL feature which can be seen in table 3.2. The feature use percentages are not consistent across the different projects, which may have various reason e.g. different use cases, different data model, varying expertise of users.

Aside from the fact that not the same feature sets have been compared, a look at the results of the LSQ analysis compared to those from table 3.1 also shows some diverging results for several features and generally higher numbers for the feature usage in the example queries from the Wikidata Query Service. This may of course be due to the fact that the example queries are intended to show off features and different use cases of the data source, which may lead to more complex queries.

Saleem et al. [SAH⁺15] also studied the shapes of the basic graph patterns based on the type of variable joins, distinguishing between star shape, sink, path, hybrid and no joins. Most commonly (66.51%), queries did not have any joins at all, followed by star shapes (33.05%). This is in line with Hernández [HHR⁺16] claiming that subject-subject joins are a common pattern.

Stegemann and Ziegler [SZ] also analyzed patterns in queries from the LSQ dataset. In order to detect patterns, they parametrized all 636,876 queries from the dataset and removed all parts that have no impact on the pattern, resulting in 1,619 unique query patterns. One major result was, that the top 12 most frequently used unique patterns that cover 85% of request in the logs were very simple. 9 out of 12 consist of a single triple pattern.

3.4 Query Builder Tools

Part of this work is to give an overview of state-of-the-art graphical user interfaces for SPARQL queries. Despite SPARQL itself being a relatively new language, there already exists quite a large number of tools, each with different goals, using different approaches, developed for different target audiences, and varying levels of work on usability. This section is intended as a general overview of a selection of these tools.

3.4.1 Overview

Query builder interfaces can be assessed based on many criteria, one of which is the required *skill level of the target audience*. On one end of the spectrum, there are projects like the Linked Data Query Wizard [HGVS14], which set out to develop a user interface based on commonly known concepts, that makes it possible for users to create SPARQL queries and explore datasets. It starts off with a keyword search, and presents the found entities as a spreadsheet, letting the user add columns based on RDF predicates and filter based on values, completely hiding the fact that the application is generating SPARQL code in the background.

A similarly novice-friendly approach is that of the ExConQuer Query Builder Tool [AOA] which, just like the Linked Data Query Wizard, hides the underlying RDF data representation. The workflow of this tool is also initiated by a keyword search, but instead of using the spreadsheet metaphor, works similar to a faceted browser, suggesting related predicates and letting the user filter the result set.

On the opposite end of the spectrum are essentially visual programming tools for SPARQL like iSPARQL [isp], SparqlFilterFlow (figure 3.2) [HLBE14] and Nitelight [RSBS08]. They cover all, or close to all features of the language and still require knowledge of the RDF graph in order for the user to produce queries.

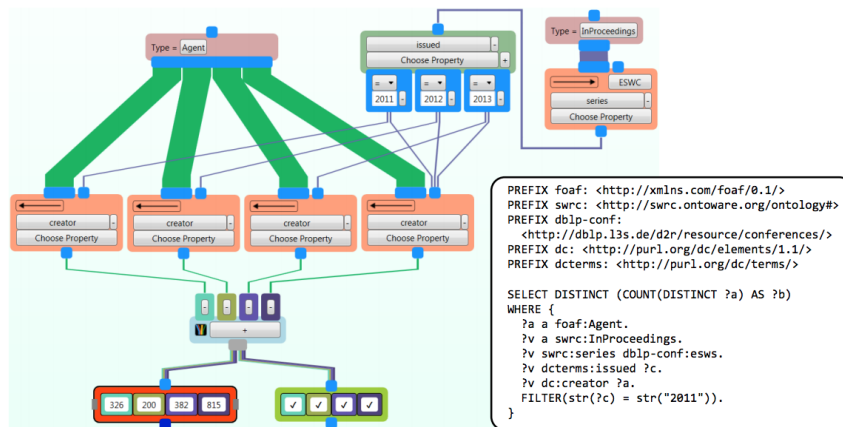


Figure 3.2: Screenshot of a query built with SparqlFilterFlow [HLBE14]

3.4. Query Builder Tools

SPARQL query builders also differ in their *support of SPARQL language features* and types of supported graph patterns. The question of what features need to be supported often goes hand in hand with the main use case for the query builder and the skill level of their target audience. Trade-offs have to be made considering whether it is more important to keep the user interface as simple as possible or supporting a certain set of features. Similarly, it might be acceptable to expose concepts like graph patterns and variables to advanced users, however, these should rather not be a major part of the user interface if the tool is targeted towards casual web users.

Query Builders that are made for advanced users like iSPARQL [isp] and Nitelight [RSBS08] show all the available features in toolbars and have lots of interface elements for constructing different kinds of queries. In the case of Nitelight, the goal was not to simplify, but to visualize structures and support experienced users in the query creation process.

The VizQuery⁴ [viz] Wikidata community project does not support any SPARQL features beyond basic triple pattern matching and in turn has a very simple interface with only a few interactive elements. As figure 3.3 shows, triple patterns are represented as an unordered list and the user can build queries with a star-shaped graph pattern by selecting properties and items. Since its interface does not provide ways to visually create more complex patterns, VizQuery allows advanced users to use variables, as seen in figure 3.4.



Figure 3.3: Screenshot of a simple VizQuery query

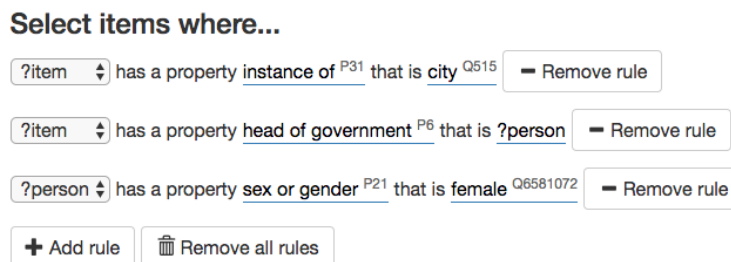


Figure 3.4: Screenshot of an advanced VizQuery query

Another distinction has to be made between query builders that are tailored to one or several specific endpoints and those that have *arbitrary endpoint*

⁴<https://tools.wmflabs.org/hay/vizquery/>

support. A GUI that works with all endpoints has its use of course, however, this also means that no assumptions about the data structure can be made and no complexity can be hidden from the user. The SPARQLGraph application for querying biological databases [STP14] has a number of different endpoints integrated and suggests relevant entities and predicates to its users as draggable UI elements, as well as sensible suggestions about how to connect entities.

Similarly, VizQuery is made specifically for Wikidata and makes use of the Wikidata API to provide auto-completion for predicates (Wikidata properties) and values (Wikidata items). The user does not need to know about prefixes for predicates and nodes and also gets labels and images automatically added to the result set.

3.4.2 Categorization

Aside from the three criteria mentioned in the previous section, one can also differentiate between the user interface paradigms that are used for the query builders. Haag et al. [HLBE14] proposed a separation between "form-based querying", "querying by browsing", and "visual query languages" which is a good starting point to further categorize related projects.

- **Form-based querying** refers to an approach that centers around text fields and step-by-step query construction that is still close to the triple pattern syntax of SPARQL. Examples mentioned for this type of query builders are Konduit VQB [AMH⁺10] and SparqlViz [BE06], both desktop applications, and the DBpedia query builder.
- **Querying by browsing** is described to be done by tools that cater to a very specific use case and do not allow the user to fundamentally change the structure of a query.
- Out of the three categories, **visual programming languages** are said to have the lowest level of abstraction on top of SPARQL and use either node-link diagrams, UML-like structures, or data flow diagrams for visualizing the query structure.

This categorization suggested by Haag et al. will be expanded upon in the next chapter, where several query builder tools will be analyzed more deeply.

3.5 Summary

Combined with the review of the theoretical foundations presented in the previous chapter, the related work that has been done around Wikipedia and lists, SPARQL analysis, and query builder tools provides the insights that are necessary for the following chapter. The overview of query builder tools will be the basis for a categorization based on user interface paradigms, and the

3.5. Summary

queries from Listeria templates found on Wikipedia are the data source for the SPARQL analysis.

4 Analysis

The previous chapter contained a brief overview of ways to analyze SPARQL queries and existing SPARQL query builder tools. In this chapter, the Listeria Bot SPARQL queries will be analyzed as part of the user research. The results of this analysis will provide insights about the types of queries that need to be supported. Combining these insights with the functional requirements from the Wikidata product management will complete the requirements elicitation process.

As a final step, state-of-the-art query builder tools will be reviewed in-depth and categorized to get an idea what main user interface paradigms are common and to draw inspiration from different projects for the user interface and interaction design. Finally, those tools that are web-based and have a working demo available online will be assessed to reflect how well each type of query builder matches the functional requirements of the Wikidata Query Builder.

4.1 Listeria SPARQL Analysis

The goal of the SPARQL analysis is to understand the needs of users that create lists with the Listeria tool. In the related literature [HHR⁺16, SAH⁺15, SZ] the focus was mainly on analyzing the use of SPARQL features and graph patterns in general. This section describes how SPARQL queries were analyzed with the objective of deriving functional requirements for the query builder.

4.1.1 Data Collection

The SPARQL queries that needed to be analyzed are spread out across pages on several Wikipedia projects, and since not all Wikipedias could be covered, the two most active ones (English and German Wikipedia [wika]), measured by the number of total edits, were chosen.

MediaWiki's `EmbeddedIn`¹ API method was used to find and download the Wikitext source of all pages that included the `Wikidata list` template. This resulted in two directories – one for each language – containing one file per found article: 681 for the German Wikipedia and 996 for the English Wikipedia.

Since the structure of the `Wikidata list` template is known and always contains a `sparql` parameter, a simple regular expression could be used to extract the SPARQL query from the Wikitext which was then written back

¹<https://www.mediawiki.org/wiki/API:Embeddedin>

4.1. Listeria SPARQL Analysis

to the text file. Out of the 682 queries from the German Wikipedia, 669 were valid SPARQL queries and 992 of the 996 English Wikipedia ones.

4.1.2 Analysis Setup

Since SPARQL is quite a complex language and can have nested structures, a simple text-based analysis was out of the question. Instead, Node.js² was set up to read the files, and the SPARQL.js³ library, that is also used for the Wikidata Query Service frontend⁴ was used to parse each query into a JSON structure that could then be traversed as a tree. The traversal was facilitated by the traverse⁵ library, and lodash⁶ was used as a utility library to allow for a more functional programming style.

The JSON structure of the "list of cats" query for example can be seen in listing 4.1.

```
1 {
2   "queryType": "SELECT",
3   "variables": [
4     "?item"
5   ],
6   "where": [
7     {
8       "type": "bgp",
9       "triples": [
10        {
11          "subject": "?item",
12          "predicate": "http://www.wikidata.org/prop/direct/P31",
13          "object": "http://www.wikidata.org/entity/Q146"
14        }
15      ]
16    }
17  ],
18  "type": "query",
19  "prefixes": {}
20 }
```

Listing 4.1: SPARQL.js output for a simple query with one variable and one triple pattern

With this structure it was rather easy to test e.g. whether a certain feature was used. A `countInQueries` helper-function was programmed to test for a boolean condition for every query within a directory, which increments a counter if the condition was true.

²<https://nodejs.org/en/>

³<https://github.com/RubenVerborgh/SPARQL.js/>

⁴<https://github.com/wikimedia/wikidata-query-gui>

⁵<https://github.com/substack/js-traverse>

⁶<https://lodash.com/>

```

1  var hasFilter = (query) => {
2      return query.traverse().reduce(function(acc, node) {
3          return acc || this.notLeaf && node.type === 'filter'
4      }, false)
5  }
6  countInQueries(EN_SPARQL, hasFilter)
7  countInQueries(DE_SPARQL, hasFilter)
8
9  // Output:
10 // data/sparql_queries_dewiki 669 51
11 // data/sparql_queries_enwiki 992 375

```

This example shows how the frequency of use of the `FILTER` feature was tested. The `hasFilter` function defines the condition that takes the parsed query as a parameter. Within the function, the `query` argument is traversed and each non-leaf node is checked, whether its type is "filter", as it would be represented in the JSON-structure of the parsed query. The `countInQueries` function then outputs for each directory, how many queries used the `FILTER` feature at least once.

4.1.3 Data Cleaning

Unfortunately, an initial analysis of SPARQL feature usage revealed that the data from the German Wikipedia is quite heavily biased as seen in table 9.3 (appendix). Upon further inspection, it appeared that two queries were duplicated with only slight modification (one Q-ID was replaced each time). The first query, causing the large number of property path usage, was duplicated 316 times, the other query, causing the many `UNION` queries was found 95 times.

In order to avoid this bias, all but one of each of the two queries were removed from the data source and the analysis run again. Since there were only 260 valid queries from the German community left, the two were analyzed as a single dataset.

4.1.4 General SPARQL Features

The following features were tested: `FILTER`, `ORDER BY`, `DISTINCT`, `BIND`, `NOT EXISTS`, `OPTIONAL`, `GROUP BY`, `VALUES`, `UNION`, `MINUS`, sub-queries, and property paths.

As figure 4.1 shows, `FILTER` is by far the most used feature, whereas sub-queries and `MINUS` are not used a lot. The observation that `ORDER BY`, `GROUP BY` and `OPTIONAL` are used less than they were in the example queries of the Wikidata Query Service that Hernández et al. [HHR⁺16] analyzed, may be due to the Listeria Bot functionality that also provides grouping, ordering and selection of optional results [wikb]. Property paths are used just as often as they are in the example queries, which, as previously mentioned, likely has to do with the way that instances and classes are modeled in Wikidata.

4.1. Listeria SPARQL Analysis

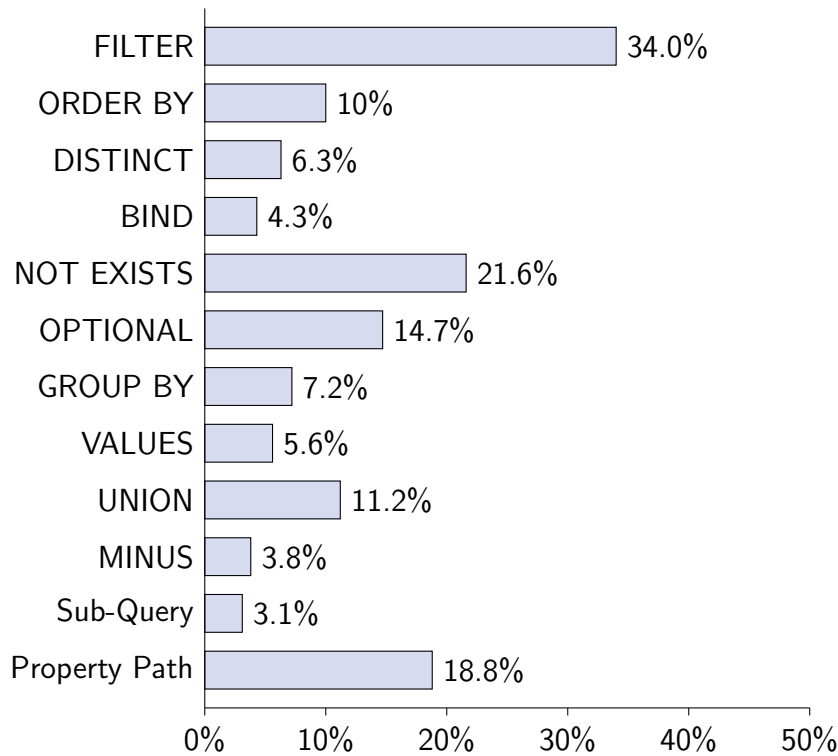


Figure 4.1: SPARQL feature usage in German and English Wikipedia Listeria queries (1252 total)

For the user interface of the query builder the minimal set of features that supports most of the queries is even more interesting than each feature considered separately. It was also considered, that some features may be more challenging to implement than others. `DISTINCT` for example, was an easy choice for a feature to be enabled for all queries by default, since the tool is meant to be used for the automatic generation of lists, and duplicate list entries are almost never desirable.

With that in mind, several combinations were manually tested and the one that supported most (76.3%) queries produced by the German and English Wikipedia community is `OPTIONAL`, `FILTER` (including `NOT EXISTS`), `ORDER BY`, `VALUES`, `DISTINCT` and property path. It should also be noted, that a manual inspection of unsupported queries revealed that many of them could be rewritten to also fit into the set of supported queries, e.g. when `UNION` was used to combine results from two triple patterns with the same subject and predicate, it might as well have used `VALUES` to combine the desired object nodes. Similarly, `BIND` was often used to parse a the Q-ID out of the item URI, which can also be integrated into the query builder as a special case, if needed.

4.1.5 Triple Pattern Structures

Prior SPARQL analyses [SAH⁺15, HHR⁺16] found that the least common type of triple pattern structure is the sink, or object-object-join. Hernández et al. [HHR⁺16] found that most graph patterns within the examples of the Wikidata Query Service follow a tree structure (no cycles and no object-object joins). The question of the triple pattern structure that supports most queries is equally important as that of what SPARQL feature need to be supported. If, in the worst case, the results show that many queries use complex structures like cycles, or hybrids of multiple join types, then the user interface would have to support these and either let the user create arbitrarily complex patterns graphically, or introduce variables like VizQuery [viz] did for advanced users. Both options would be highly questionable from a usability standpoint since the tool is meant to be for people without prior SPARQL experience.

As done in the work of Saleem et al. [HHR⁺16], the basic graph patterns in the queries were inspected for subject-subject, subject-object, and object-object joins. Additionally, cycles (paths that visit any node more than once) were added to the list in order to potentially confirm the assumption that most queries can be represented as trees.

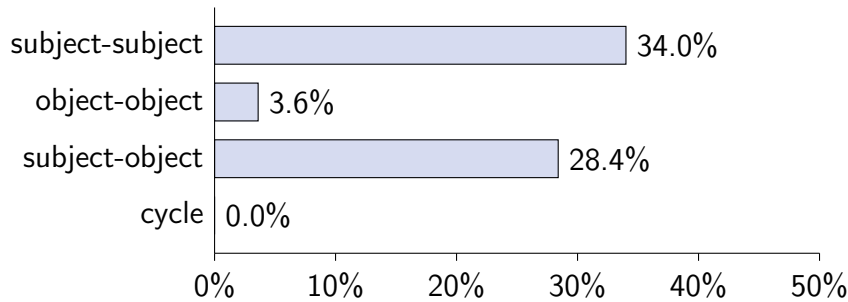


Figure 4.2: Graph patterns in SPARQL queries of the German and English Wikipedia Listeria queries (1252 total)

As shown in figure 4.2, subject-subject and subject-object joins are indeed the most common patterns. Since object-object joins only make up 3.6% and cycles are non-existent, it can be assumed that a tree structure will be sufficient to represent a large majority of the queries.

4.1.6 Common Property Paths

As mentioned before, property paths are syntactically similar to regular expressions and can be rather complicated, which makes full support for arbitrary property path patterns problematic for a user interface that aims to be as simple as possible. Personal experience and the work from Hernandez et al. [HHR⁺16] suggest that there exists a small subset of reoccurring property paths that cover most cases. The one mentioned frequently is "instance

4.1. Listeria SPARQL Analysis

of any subclass of” (`wdt:P31/wdt:P279*`), another candidate may be the recursive location lookups (“a place located in a place, located in a place...” – `wdt:P131*`).

The exact combination of `wdt:P31/wdt:P279*` appeared in 20% of the 235 queries with property paths, any combination of `wdt:P279*` in 60.4%, and `wdt:P131*` in 51.5%. For a more general overview, figure 4.3 shows the most common properties used in property paths. If the same property occurred multiple times in one query or even in one property path, it was also counted multiple times.

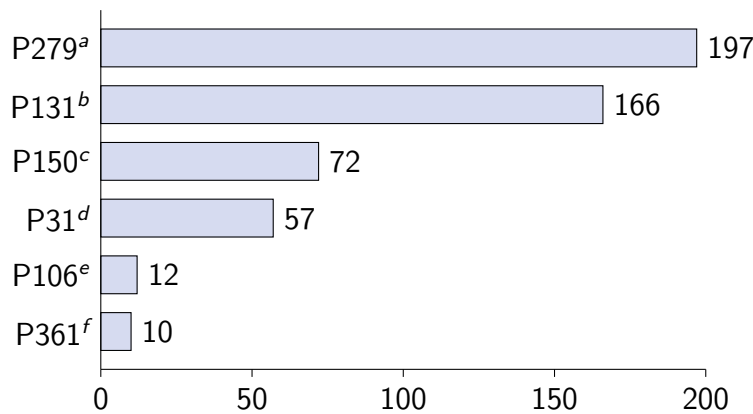


Figure 4.3: Most commonly occurring properties in property paths in SPARQL queries of the German and English Wikipedia Listeria queries; Showing only those with 10 or more occurrences

^asubclass of

^blocated in the administrative territorial entity

^ccontains administrative territorial entity

^dinstance of

^eoccupation

^fpart of

Within the 235 queries that use property paths, 541 properties were used in total as part of property paths, but only 22 different ones. Out of those 22, 4 properties make up over 90%. This suggests that in order to support the large majority of property paths used in those queries, not all possible combinations have to be supported, but only the most common ones.

4.1.7 Commonly Used Wikidata Features

As a final step of the SPARQL analysis the commonly used Wikidata-specific features were analyzed with the goal of finding out which ones are included frequently in queries and can be abstracted in some way. For querying items, the so called “main snak” of the statement (the property-value-pair without qualifiers and references), was already considered a must-have feature for the

query builder interface, so only *statement qualifiers*, *statement references*, and *sitelinks* were analyzed.

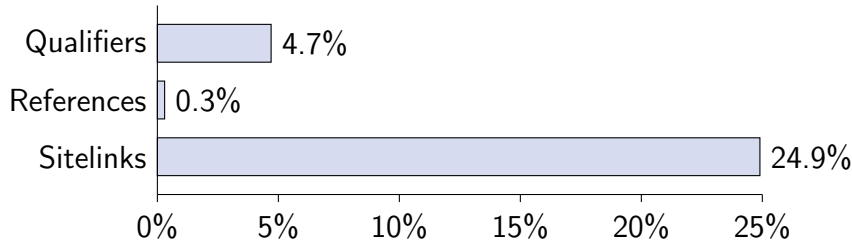


Figure 4.4: Usage of Wikidata-specific features in German and English Wikipedia Listeria queries (1252 total)

As shown in figure 4.4, sitelinks are used quite often, which makes sense, since especially links to other articles are highly relevant for lists within Wikipedia. References are barely used, found in only 4 out of the 1252 queries. Qualifiers were not used a lot, although it is suspected that this might be due to their complicated modeling in the Wikidata RDF schema.

4.2 Requirements

With the gained insights from the Listeria SPARQL query analysis, the requirements for the query builder were gathered together with Wikidata’s product manager. The requirements were framed as attributes in order to reuse them for the detailed assessment of related query builder tools. Attributes were categorized as *feature-specific*, *interface-specific*, and *interaction-specific*.

4.2.1 Feature-specific Attributes

Feature-specific attributes are those that describe the capability of the tool. The initial set of feature-specific attributes are the set of SPARQL features and basic graph patterns that is able to build most of the Listeria queries, taken from the SPARQL analysis in the previous section. The additional features came from the product management.

Attributes **A1-A8** are derived from the analysis of the previous chapter. The ability to show the results of the query in tabular form (attribute **A9**) are important since the user likely wants to see what elements are contained in the list that the query produces. Since the query is meant to be exported and used on Wikipedia, the query export feature (attribute **A10**) is equally important. The last feature is about being able to share, import and export a query (attribute **A11**). This is considered important because according to experts and people with access to the logs of the Wikidata Query Service, many users of the SPARQL frontend write SPARQL queries by modifying the example queries.

4.2. Requirements

Code	Attribute	Scale
A1	Supports FILTER	Yes/No
A2	Supports OPTIONAL	Yes/No
A3	Supports VALUES	Yes/No
A4	Supports NOT EXISTS	Yes/No
A5	Supports ORDER BY	Yes/No
A6	Supports property paths	Yes/No
A7	Supports subject-subject joins	Yes/No
A8	Supports subject-object joins	Yes/No
A9	Shows results of the query in tabular form	Yes/No
A10	Allows export of the query as text	Yes/No
A11	Allows import and export of example queries	Yes/No

Table 4.1: Feature-specific attributes

4.2.2 Interface-specific Attributes

Interface-specific attributes affect the usability of the solution, in other words, they determine how well the user can operate the interface to utilize the features described in the previous section. They describe structural features of the user interface.

Code	Attribute	Scale
A12	Main interface paradigm	Nominal
A13	Simplification through SPARQL language abstractions	Yes/No
A14	Simplification through domain-specific abstractions	Yes/No

Table 4.2: Interface-specific attributes

The main interface paradigm (attribute **A12**) is mainly interesting for the assessment following in the next section and will show a possible relationship between the type of interface and the attributes it possesses. The main interface paradigm will be assessed on a nominal scale with categories that will be determined in the following section as well.

Since the goal is to create a tool for non-experts, it is important that some part of the query building process is simplified through the user interface. Attribute **A13** is targeted towards simplifications concerning SPARQL itself, whereas attribute **A14** is about abstractions that are specific to the domain or RDF schema. This, of course, is only possible for query builders not supporting arbitrary endpoints.

4.2.3 Interaction-specific Attributes

Interaction-specific attributes describe, as the name suggests, behavioral properties of the user interface that concern user interaction. In addition to auto-

completion for RDF entities such as predicates and nodes (attribute **A16**), it was also considered useful to measure the number of user interactions (measured in clicks and keystrokes) it takes to build a simple query made of two triple patterns.

Attribute **A17** describes the context-awareness of the interface, e.g. in the situation that the user entered "country" as the predicate of a triple pattern and then intends to fill in a value for the object. In this case, a context-based suggestion would be a list of countries as possible values for the object.

Code	Attribute	Scale
A15	Auto-completion for entities	Yes/No
A16	Context-based suggestions	Yes/No
A17	Interaction effort to build a simple query	Click/keystroke count

Table 4.3: Interaction-specific attributes

4.3 State-of-the-Art Review

Section 3.4.1 gave a brief overview of query builder tools and their differences based on the skill level of their target audience, whether they support specific or arbitrary SPARQL endpoints, the support for SPARQL language features and the user interface paradigm. The first two of the criteria are already determined by the goal of this project. The target audience are Wikidata users who know the data model, but not SPARQL, and it only needs to support Wikidata's own SPARQL endpoint. This section's goal is to categorize query builder tools based on their user interface paradigm.

4.3.1 Categories

Haag et al. [HLBE14] proposed the categories *form-based querying*, *querying by browsing* and *visual programming languages* and found several representatives for each group, however, the category names are rather generic and do not immediately reveal the linked data and SPARQL context. Also, the fact that form-based querying approaches and visual programming query builders both have triple pattern based query building at their core might suggest that the two are closely related.

Based on these observations, the following hierarchical categories are proposed as an alternative:

- Exploratory Query Builders
 - Faceted Search
 - Spreadsheet
- Triple Pattern Based Query Builders

4.3. State-of-the-Art Review

- Graph
- Filter/Flow
- Tree
- Unstructured

The first category of *exploratory query builders* describes largely the same as the *querying by browsing* paradigm proposed by Haag et al. [HLBE14], with the additional distinction between faceted search and spreadsheet interfaces. The second group describes query builder interfaces that utilize triple patterns at their core, differentiating between interfaces that present graph patterns as graphs, trees, or with no fixed structure. The aforementioned *visual programming languages* mostly fall into the category of *triple pattern based query builders* with *graph* structure. *Form-based* step-by-step query builder interfaces are found to either use tree structures to represent queries or unstructured lists of triple patterns.

An overview of all categorized SPARQL query builders in alphabetical order can be seen in table 4.4. Category names are abbreviated with the initial letters, e.g. EF for exploratory query builder using the faceted search paradigm. The same table containing the full URLs can be found in the appendix in table 9.1.

Project Name	Category	Web Demo Available	Source
ExConQuer	EF	-	[AOA]
Freebase Parallax	EF	-	[HK09]
iSPARQL	TG	✓	[isp]
Konduit VQB	TT	-	[AMH ⁺ 10]
LDQW ^a	ES	✓	[HGVS14]
NITELIGHT	TG	-	[RSBS08]
QueryVOWL	TG	✓	[HLSE15]
Sparklis	TT	✓	[Fer14]
SparqlFilterFlow	TF	✓	[HLBE14]
SPARQLGraph	TG	✓	[STP14]
SPARQLViz	TU	-	[BE06]
Visual SPARQL Builder	TG	✓	[Eip15]
VizQuery	TU	✓	[viz]

Table 4.4: Assessed query builder tools

^aLinked Data Query Wizard

4.3.2 Exploratory Query Builders

With exploratory query builders the creation of the query happens implicitly as a side-effect of the user browsing or exploring the data graph [HLBE14]. Three

of such interfaces were examined: the Linked Data Query Wizard [HGVS14], ExConQuer [AOA], and Freebase Parallax [HK09].



Figure 4.5: Initial search in Freebase Parallax (picture from [HK09])

All three tools start off with text search as the first step and then let the user expand or filter the results. For the initial search, ExConQuer and Parallax both automatically suggest concepts and entities to the user based on the search term (figure 4.5), whereas the Linked Data Query Wizard performs a full-text search.

The second step is where their interfaces differ. Freebase Parallax allows the current result set to be expanded by browsing "connections" e.g. if the current results are the presidents of the United States of America, connections might be their children, spouses, or parties they are associated with [HK09].

Both Freebase Parallax and ExConQuer filter results in a way that is similar to faceted search. Faceted search [Tun09] allows users to progressively elaborate their search based on attributes (facets) of the possible results. In ExConQuer [AOA] and Freebase Parallax [HK09], these facets are based on common predicates connected to the nodes in the result set. An example of facets in ExConQuer can be seen in figure 4.6.

4.3. State-of-the-Art Review

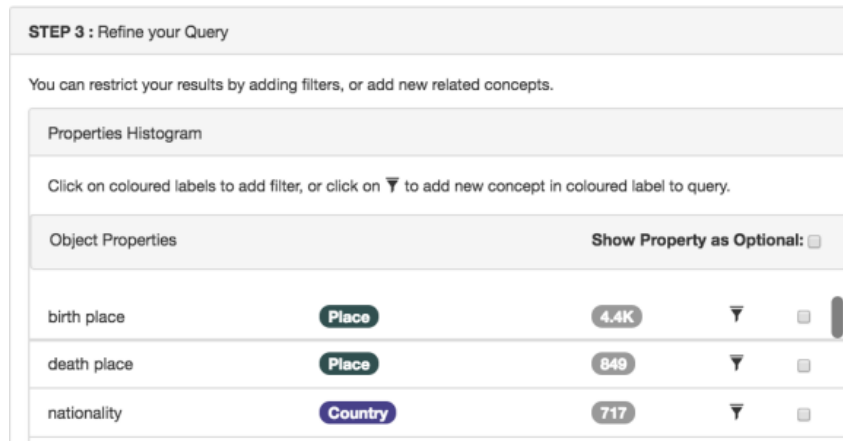


Figure 4.6: Faceted search in ExConQuer (picture adapted from [AOA])

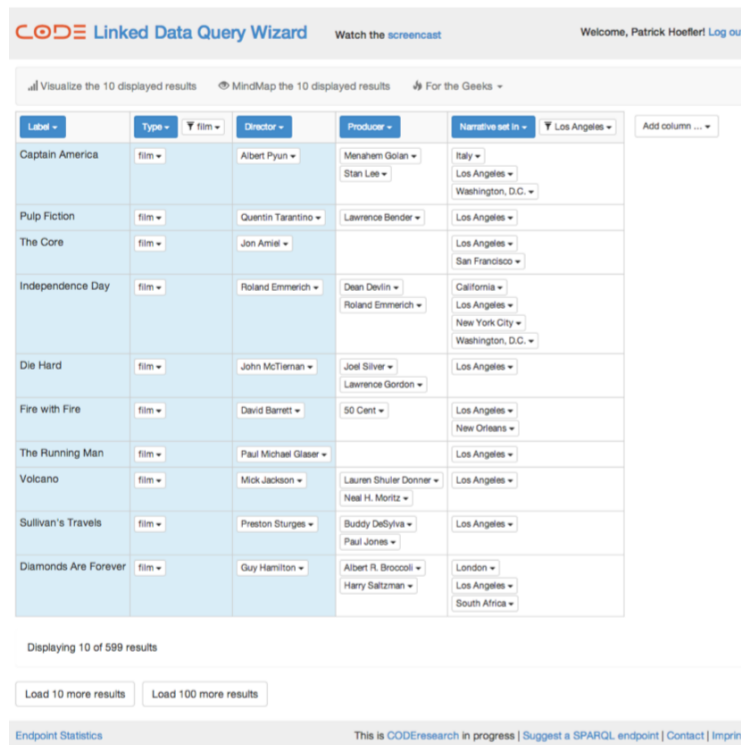


Figure 4.7: Linked Data as a spreadsheet (picture adapted from [HGVS14])

With the Linked Data Query Wizard [HGVS14], data is displayed as a spreadsheet after the initial search, with a "label" and a "type" column by default. Each row in the spreadsheet corresponds to a subject in the result set, each column to a predicate, and each cell to the corresponding object of the subject-predicate pair. The results can be expanded by adding new columns, which are suggested, similar to the facets in the other two projects, based

on common predicates of the result nodes. Filters can be set for each predicate in a way that is intended to be familiar from working with spreadsheet applications. An example of this can be seen in figure 4.7

To summarize, the key attributes of exploratory query builders are

- begin with initial search
- result-based filtering
- context-based suggestions for related concepts
- implicit query creation

4.3.3 Triple Pattern Based Query Builders

As opposed to exploratory query builders, those based on triple patterns have a more explicit way of creating SPARQL queries. Instead of the textual **subject predicate object** . notation, tools within this category provide some structure or visualization for the user to create the query's graph pattern.

Triple Pattern Based Query Builders with Graph Structure

This was found to be the most common category of query builder tools, which corresponds to the one that Haag et al. [HLBE14] called visual programming tools. While they differ in many regards, each of the tools provided the user with a canvas and graphical elements for nodes that can be connected with predicates to visually create the graph pattern, as seen in figure 4.8.

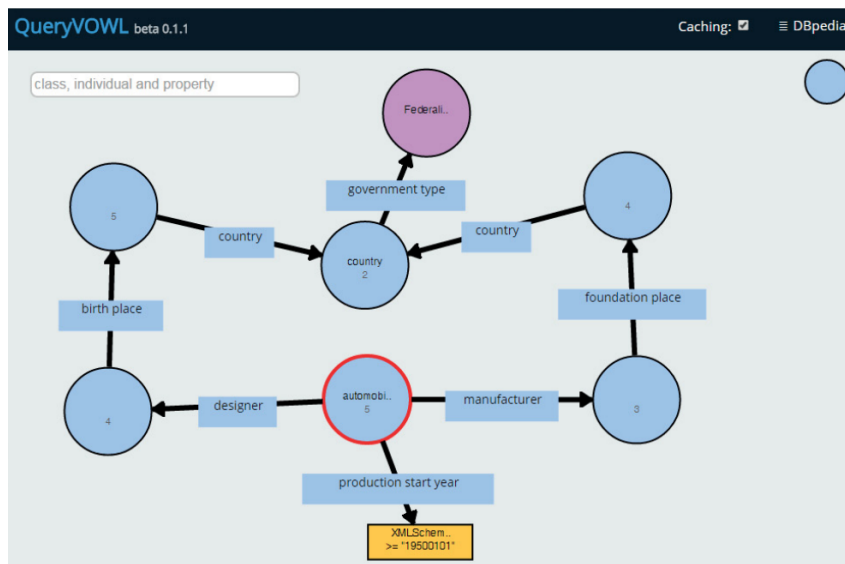


Figure 4.8: SPARQL query as a graph (picture adapted from [HLSE15])

Triple Pattern Based Query Builders with Filter/Flow Structure

The only tool that was examined that uses the filter/flow paradigm is SparqlFilterFlow [HLBE14]. Similar to query builders with graph structure, this one also required the user to graphically build the query structure by elements through a drag and drop interface, however, instead of the RDF like graph structure with subjects and objects as nodes, and predicates as vertices, in SparqlFilterFlow the nodes are triple patterns and filters, and vertices visualize the data flow.

Triple Pattern Based Query Builders with Tree Structure

In contrast to query builders that presented the query's graph pattern as an actual graph, query builder that utilized a tree structure as their query representation did not require the user to drag and drop vertices and edges of the graph by hand, and instead chose a form-based approach with text-fields as input elements, and indentation as a means to visualize the parent-child relationship of tree nodes.

Figure 4.9 shows how Konduit VQB [AMH⁺10], a desktop application, uses indentation to visualize the graph pattern as a tree, with the root subject on the top, and predicate-object-tuples on every following line that make up the pattern. Note that the subject depends on the level of indentation e.g. the first and second predicates "name" and "publication" belong to the person, whereas the further indented "topic" has the publication as the subject.

In addition to that, there is no obvious distinction between triple patterns and FILTERs, with the exception that in addition to a predicate and an object, a filter function is specified. As seen in the query on the bottom of figure 4.9, the `name contains "K"` was mapped to one triple pattern and one FILTER-expression.

Another example of a query builder tool that represents its query's graph pattern as a tree is Sparklis [Fer14]. It is different from Konduit VQB in that it is a web-based application and takes a more natural language oriented approach, however, despite the fact that the author claims to have taken inspiration from faceted search interfaces, it does have the explicit query building workflow and it also uses indentation to visualize the subject context as shown in figure 4.10.

Unstructured Triple Pattern Based Query Builders

One of the tools that fall into this category is Wikidata VizQuery [viz]. As previously described, it lets users specify simple star-shaped graph patterns as a list and requires the use of variables to form more complex patterns.

The other tool is SPARQLViz [BE06], a desktop application with a wizard-like workflow, that allows the users to input arbitrary triples in any order without much abstraction from the SPARQL language.

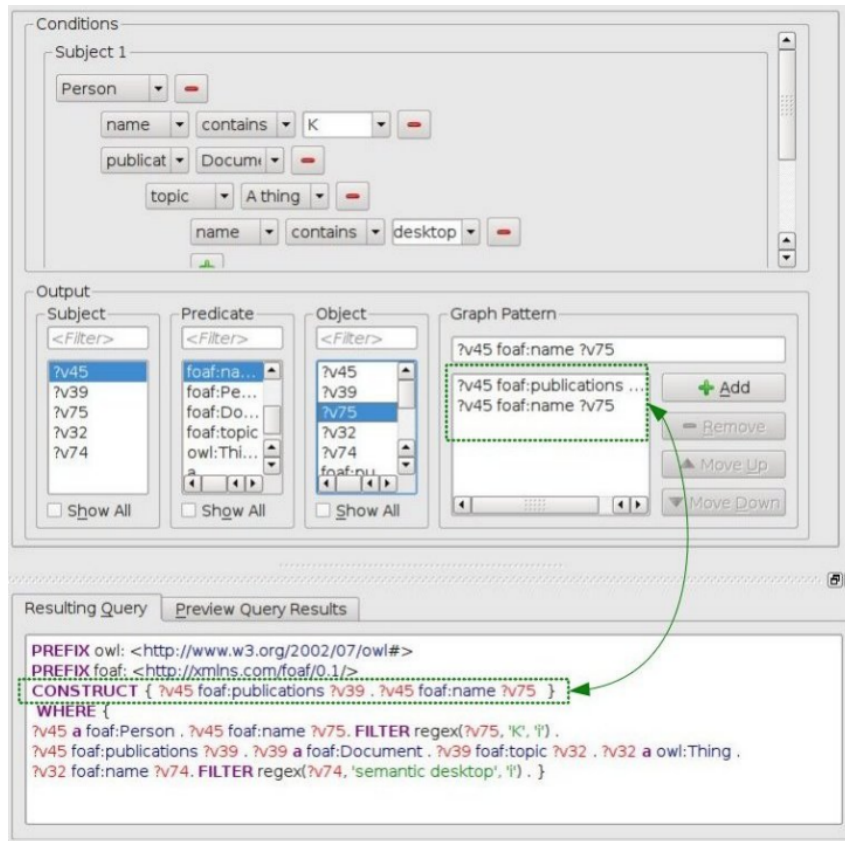


Figure 4.9: SPARQL query as a tree (picture adapted from [AMH⁺10])

give me every band
 whose genre is Rock music
 and whose hometown is
 Australia
 or anything whose country is Australia

Figure 4.10: Screenshot of Sparklis [Fer14] from <http://www.irisa.fr/LIS/ferre/sparklis/>

4.3.4 Attribute-based Assessment

With the categories and desirable attributes (from section 4.2) of query builders defined, an assessment of query builders based on these attributes is presented in this section. The goal is to find out which category of query builders is best suited for the requirements of the Wikidata Query Builder.

From the list of query builders in table 4.4, those with a web demo available were chosen for this assessment. For the SPARQL feature attributes (A1-A8) the attribute was rated with "yes", if the feature was supported to some extent, e.g. if some, but not all FILTER-functions were possible. If not further specified,

4.3. State-of-the-Art Review

the query that is built for the interaction-specific attribute for testing the interaction effort for a simple query (**A17**), will consist of two triple patterns to answer the question "Which movies were directed by Tim Burton and had Helena Bonham-Carter as part of their cast?".

The eight chosen query builder tools will be individually described, and notable properties pointed out. At the end of this subsection, table 4.5 summarizes the results. Screenshots of the queries built for the interaction effort evaluation can be found in the appendix.

iSPARQL

iSPARQL, which is used as a frontend for the DBpedia SPARQL endpoint⁷, is a query builder tool for advanced users featuring the typical drag and drop interface for graph nodes and predicates. Its default screen is the textual query builder and only a click on the button labeled "Visualize" leads to the graphical query builder. It supports all the feature-specific attributes, however, the user interface offers very little simplification other than the visualization of the query's graph pattern, and contains many toolbars, buttons and controls to accommodate all of SPARQL's functionality.

Linked Data Query Wizard

The Linked Data Query Wizard aims to "dramatically simplify" querying and accessing data with SPARQL by letting its users explore the RDF graph utilizing features of commonly known concepts like full text search and spreadsheets [HGVS14]. It supports star-shaped graph patterns by selecting filters for predicate columns, but no combination of star-shapes and path-patterns is possible, because of the two-dimensional nature of spreadsheets.

Almost all interactions with the data are done by clicking, including adding filters and adding new columns for predicates. The predicates are presented in a long, alphabetically ordered dropdown-list (figure 4.11, which makes finding the desired predicate cumbersome in some cases.

Due to this mouse-oriented interface, the interaction effort was lower than in other tools, however, the implicit query building approach required a very indirect path to get the desired result. Since both predicates "film director" and "starring" are directed from the film to the person in the RDF graph, it was not possible to get from either "Tim Burton" or "Helena Bonham-Carter" to the list of films directed by Tim Burton starring Helena Bonham-Carter. Instead, one specific film title was entered in the initial search field, then the "film director" and "starring" columns were added, the director and actress chosen as filters, and then the film title was removed as a filter.

⁷<http://dbpedia.org/isparql/>

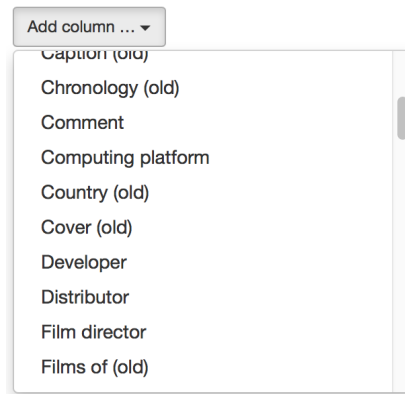


Figure 4.11: Screenshot of the Linked Data Query Wizard's interface element for adding predicates

QueryVOWL

QueryVOWL, like iSPARQL, represents the query pattern as a graph and lets the users connect nodes with predicates. Since it has features like auto-completion for nodes and predicates, and even only shows relevant predicates (context-based suggestions), so it feels more user-friendly. On the other hand, the generation of the suggestions often takes a long time and completely blocks the interface. Like the previous two, it does not provide any domain specific simplifications since it is intended for arbitrary SPARQL endpoints.

Sparklis

Compared to the other projects, Sparklis takes a much more natural language oriented approach. The query is represented as a imperative sentence in a tree structure and the user can focus different parts of the sentence (subject, predicate and object) to set the context for the other user interface elements to refine the query, as it can be seen in figure 4.12.

The suggestions to refine the query are divided into "*concepts*", which include predicates and classes of things, "*entities*", which are concrete things (people, artworks, places, etc.) and "*modifiers*", which can be used to combine and modify values. As with QueryVOWL, the suggestions for these three user interface elements depend on the query that is being constructed and on the part of the query that is focused. Unfortunately, it has the same problem of the slowly updating interface while the suggestions are being generated.

SparqlFilterFlow

SparqlFilterFlow is another very powerful query builder tool that, like iSPARQL and QueryVOWL, has a more visual query building process, but instead of the graph representation uses the filter/flow paradigm. It does offer auto-completion of nodes and predicates, however, the interface completely

4.3. State-of-the-Art Review

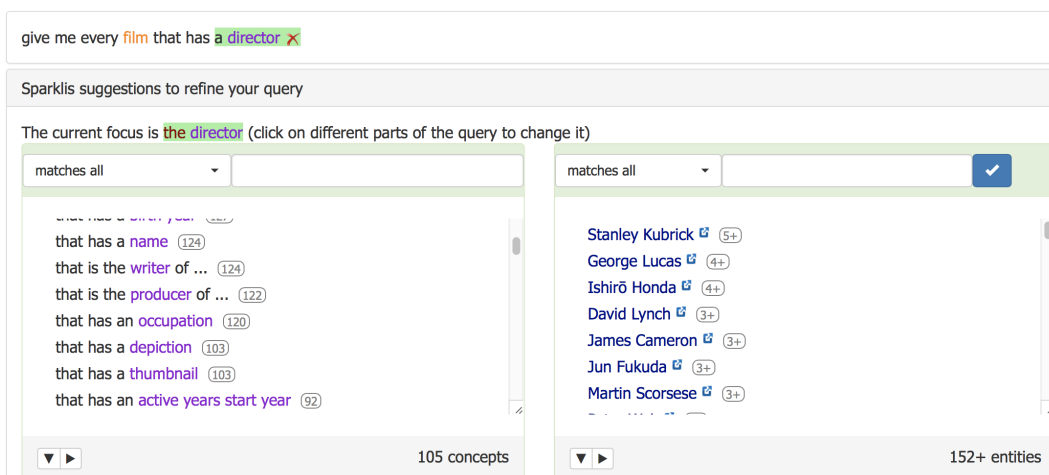


Figure 4.12: Sparklis query with focus on the predicate of a triple

hung up for several minutes whenever larger result bases (like actors) were searched, so that the query had to be changed to "Countries were French is spoken that have Euro as their currency".

Another problem is that the web version⁸ is only a "lightweight" web demo of the one described in the original paper [HLBE14]. This demo did not appear to support `FILTER`, `VALUES`, `OPTIONAL` or property paths, and also did not show the textual version of the query that was built.

SPARQLGraph

SPARQLGraph is one of the few query builders that were designed for a specific set of SPARQL endpoints of RDF databases for biological data. It provides interface elements for types of nodes that can be dragged onto the query builder canvas and also includes predefined predicates to connect the nodes. Unfortunately, the query execution seemed to be no longer functional, since neither the example queries, nor manually built queries produced any results, so that no effort analysis could be done.

Visual SPARQL Builder

The Visual SPARQL Builder [Eip15] is limited to connecting nodes with variables and does not provide any way of specifying triple patterns with specific entities (URIs) as objects. In order to assess the interaction effort, the query "People that were born in the same place they died" (figure was chosen as an alternative. Aside from its limitations, the user interface is somewhat similar to QueryVOWL and also provides auto-completion for predicates and concepts. The only type of filter it supports is the `NOT EXISTS` function.

⁸<http://sparql.visualdataweb.org/sparqlfilterflow.php>

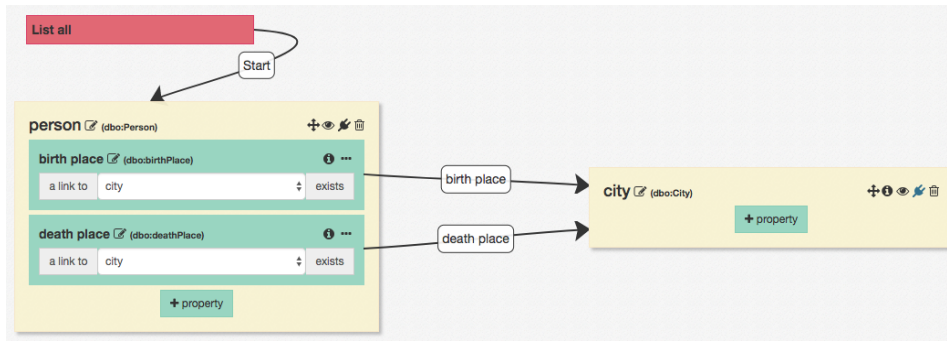


Figure 4.13: Screenshot of the query built using the Visual SPARQL Builder to assess the interaction effort

VizQuery

VizQuery was built for Wikidata and, without using variables, only allows specifying queries with star-shaped graph patterns. It uses Wikidata’s API for auto-completing predicates (Wikidata properties) and values (Wikidata items) but does not support any SPARQL features other than triple patterns.

Summary

The detailed analysis has shed some light on the strengths, weaknesses, and trade-offs of approaches that the different tools take. Text-based tools like Sparklis are limited by their query representation and cannot represent cyclic graph patterns, however, more powerful alternatives require the user to visually sketch out the graph or use variables. Auto-completion for text-fields improves the user-friendliness of the tools and could be improved by only providing context-based suggestions, which in turn often come at the cost of performance.

Table 4.5 shows the results of the assessment. Out of the 8 assessed tools, Sparklis has the best support for the defined attributes. It covers all desired SPARQL features except property paths, and also provides example queries, which many other query builders did not. The main issues are that there is no user study showing that it is suited for non-expert users, and if it were to be used with Wikidata, it would need additional abstractions for statement qualifiers in order to be useful.

The most commonly supported attributes are the results view, support for basic graph patterns, and query export. The least frequently supported attributes are property paths and domain-specific abstractions.

4.4 Summary

The analysis of Listeria SPARQL queries from the German and English Wikipedia, and analysis of related projects have provided the answers to all the questions that came up in the beginning of the design process of the query builder. By

4.4. Summary

	iS ^a	LDQW ^b	QV ^c	Sparklis	SFF ^d	SG ^e	VSB ^f	VQ ^g
Supports FILTER (A1)	✓	✓	✓	✓	-	✓	✓	-
Supports OPTIONAL (A2)	✓	✓	-	✓	-	-	✓	-
Supports VALUES (A3)	✓	-	-	✓	-	-	-	-
Supports NOT EXISTS (A4)	✓	-	-	✓	-	-	-	-
Supports ORDER BY (A5)	✓	-	-	✓	-	-	-	-
Supports property paths (A6)	✓	-	-	-	-	-	-	-
Subject-subject joins (A7)	✓	✓	✓	✓	✓	✓	✓	✓
Subject-object joins (A8)	✓	-	✓	✓	✓	✓	✓	✓
Tabular results (A9)	✓	✓	✓	✓	✓	?	✓	✓
Query Export (A10)	✓	✓	✓	✓	-	✓	✓	✓
Example queries (A11)	-	-	-	✓	-	✓	-	✓
Main interface paradigm (A12)	TG	ES	TG	TT	TF	TG	TG	TU
SPARQL simplification (A13)	-	✓	-	✓	✓	✓	-	✓
Domain-specific simplification (A14)	-	-	-	-	-	✓	-	-
Auto-completion for entities (A15)	-	-	✓	✓	✓	-	✓	✓
Context-based suggestions (A16)	-	✓	✓	✓	✓	✓	-	-
Interaction effort (A17)	84	26	35	29	59	-	40	33

Table 4.5: Query Builder Tools Assessment

^aiSPARQL

^bLinked Data Query Wizard

^cQueryVOWL (Web Demo)

^dSparqlFilterFlow (Web Demo)

^eSPARQLGraph

^fVisual SPARQL Builder

^gVizQuery

understanding the structure and SPARQL feature usage of the Listeria queries, trade-offs between usability and support for certain SPARQL features can be considered and decisions made deliberately. The analysis and categorization of other query builders yielded insights about what approaches exist, and how certain features may be adapted in the development of the Wikidata Query Builder.

The central discoveries of the SPARQL analysis are the set of SPARQL features and graph patterns that the tool needs to support. Property paths are likely used more frequently in Wikidata queries because of the way that instances, classes, and locations are modeled, but only a limited number of properties is frequently used, so that it may be sufficient to handle specific cases. Subject-subject and subject-object patterns are the most common graph patterns seen in Listeria queries, whereas cycles and object-object joins are rarely found. Of the concepts related to the Wikidata data model, references are almost never used in queries, whereas qualifiers appear occasionally and sitelinks very frequently.

The key insights gained by the categorization and analysis of related tools are that there are two main categories – *exploratory* and *triple pattern based* query builders. Exploratory SPARQL builders are probably not ideal for directed query building, but they tend to be more user-friendly. Triple patterns based SPARQL query builders are closer to the SPARQL language itself and use varying levels of abstraction and support for language features.

4.4. Summary

5 Design

This chapter describes the design solution that was developed to meet the requirements which resulted from the user research presented in the previous chapter. The design concept, based on the ideas from related projects combined with concepts and vocabulary of the Wikidata community, will be presented in the first section. The following two sections describe the iterative development of the low-fidelity and high-fidelity prototypes, and how usability testing methods were applied to continuously improve the design of the Wikidata Query Builder.

5.1 Conceptual Model

A *conceptual model* is a high-level description of the central ideas, concepts, and elements of a design. The *mental model* of the users are their expectations about how the system and its elements work, and what consequences to expect from certain actions [Nor13, p. 13]. Without a user centered design approach, it is often the case that a designer creates a conceptual model that reflects their own mental model, which is not always the same as their users', especially when a lot of background knowledge is required to fully understand the system. To ensure good usability from the start, the conceptual model of the user interface should be as close as possible to the users' mental model.

In the context of this project, the users are a subgroup of the Wikidata community and likely expect a list generation tool to use concepts that they know from Wikidata. Concepts like triple patterns, graph structures, **FILTER** and property path, that appeared in the previous section about query builder interfaces are not necessarily part of a Wikidata user's vocabulary. In order to present intuitively understandable concepts, it is important that the language of the interface matches that of the user. In this case, the *principle of mapping* [Nor13, p. 23] can be applied.

In "The Design of Everyday Things" [Nor13], mapping is described at the example of car controls. The example of the steering wheel is somewhat special, because it describes a *natural mapping* – the physical analogy between moving the steering wheel clockwise (top moves right) to turn the car to the right. A similar physical analogy would be the movement of a mouse pointer on a computer screen.

Although not physical analogies, there exist ways to map the SPARQL querying of Wikidata's SPARQL endpoint to concepts that are known to people familiar with Wikidata's data model. In the following sections, each mapping will be introduced with a short description and excerpts from the low-fidelity

5.1. Conceptual Model

prototype.

Statement Filters

As mentioned in section 2.3 about the data model of Wikidata, statements are the key to how knowledge about things is modeled, and so they will also be central to the interface for querying. The idea of this mapping is to replace SPARQL’s subject-predicate-object notation with item-property-value and replace the term ”pattern matching” with the more commonly known ”filter”. The concept of filtering should be familiar to the large majority of users from advanced search interfaces and faceted search interfaces, e.g. from e-commerce websites.

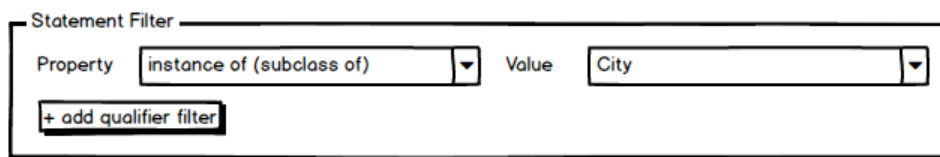


Figure 5.1: Statement filter with property ”instance of (subclass of)” and value ”City” to find all city-items in Wikidata

The goal of these statement filters is to combine several concepts from SPARQL into a single user interface metaphor that lets the user create expressive filters while hiding the underlying complexity. Each statement filter is made of two components – the property selector, and the value selector.

Property Selector

The property selector is a type-ahead input field suggesting Wikidata properties based on the user’s input. It contains some additional behavior to accommodate the frequently occurring property path expressions that were discussed in the analysis chapter. Property paths are represented as ”special properties” that appear in the suggestions alongside their connected properties¹, so when a user would type ”instance of” into the property selector field, both the plain ”instance of” suggestion would appear, as well as the ”instance of (subclass of)”. In the sense of mapping, the property selector represents the predicate, and property paths are hidden behind special properties.

Value Selector

In the simplest case, the value selector is to the statement filter, what the object is to a triple pattern in SPARQL. It also encapsulates additional behavior to simplify constructs with `FILTER` and `VALUES`.

¹This was changed in a later iteration, since the special properties were often ignored or not understood.

The behavior of the "value" field depends on the chosen property's data type. The following cases were considered for the initial user interface:

- **Data type "item"** (e.g. instance of): If the data type is "item", then the main functionality of the value field is an item selector to select specific items as the statement filter's value, as shown in figure 5.1, which would result in a single triple pattern of `?item wdt:[property-ID] wd:[item-ID] .`, where "[property-ID]" and "[item-ID]" are replaced by the IDs of the entities that the user selected. Additionally, the value field would show a drop-down² containing special values:
 - **any item** with the description *"The value does not matter as long as there is a statement for this property"*: This option would result in a triple pattern like `?item wdt:[property-ID] ?uniqueVariable .`, where "?uniqueVariable" would be replaced with a unique variable name within the query.
 - **any item matching...** with the description *"Specify possible values with statement filters"*: This option created a new level of indentation and offered the user the possibility to describe values that matched another set of statement filters – thereby representing a path in the query's graph pattern. The label "any item matching..." was found to create more natural sounding structures like "Items that are instance of city with a statement for country, that has a value of any item matching the statement continent: Europe"
 - **does not exist** with the description *"A statement for this property must not exist"*: This results in a `FILTER NOT EXISTS` for the subject and property.
 - **multiple items** with the description *"Specify multiple possible item values for this property"*: This creates a single triple pattern with group of items using the `VALUES` feature for the object.
- **Data type "time"** (e.g. date of birth): For date-properties, the value field would contain one select field for choosing the type of function such as "exact date", "before", "after", "between", and either one or two date pickers depending on the function, creating a single triple pattern for "exact date", and one triple pattern and one `FILTER` for the other functions.
- **Data type "string"** (e.g. first name): Similar to date, this would have one function for exact matches and one describing a contains-relation.
- **Data type "quantity"** (e.g. population): Again similar to the other two literal values, this would show a select field for "equal to", "less than",

²This was changed later on, as such a mix of functionality for a single user interface element was found to be confusing.

5.1. Conceptual Model

”greater than” and ”between” and one or two fields for the input of integer values depending on the function.

Each of the literal data types (time, string, quantity) also get a ”does not exist” option, which has the same effect as the one for item properties.

Multiple statement filters on the same level form a star shaped query pattern and chained statement filters using the ”any item matching...” special value (figure 5.2) form a path shape. A combination of the two enables building tree shaped graph patterns which is the minimal requirement that was the outcome of the analysis chapter.

Property Value

Property Value

Figure 5.2: Two nested statement filters; This query would result in a list of items of female government leaders

Qualifier Filters

Just like statements can have qualifiers in Wikidata, statement filters can have qualifier filters in the query builder interface. Qualifier filters behave exactly the same way as statement filters with the same functionality for property-value pairs, including the special values. The only exception is that qualifier filters cannot have further qualifier filters.

Statement

Property Value

Qualifiers

Property Value

Figure 5.3: Statement filter with a qualifier filter; This query would find any item that has a head of government with any value, and no qualifier for end date (i.e. that is the current head of government)

Result Columns

Result columns are a means to add values of properties to the resulting list. The use of the word ”column” is in line with the `columns` parameter in the template of the Listeria tool [wikb]. The user interface element that was chosen

for this functionality is another input field with type-ahead functionality for selecting properties. Each of the selected properties would create a triple pattern wrapped in an `OPTIONAL`, which again is equivalent to the functionality of Listeria's `columns` feature.

List Options

The list options map the functionality of `LIMIT` and `ORDER BY` to user interface elements. `LIMIT` can be set by either checking the "show all results" checkbox, or entering a fixed number in a text field. `ORDER BY` is realized as a select field that contains all property labels that have a variable value field in the query.

5.2 Low-Fidelity Prototype

As mentioned in the previous section, it is often the case that user interfaces are designed according to the designer's or an expert's mental model, and even a design based on best practices and guidelines does not equate to good usability [LFH17, p. 915]. Formative evaluation offers a relatively low-effort and quick way to ensure that the rough structure of the user interface is usable by iteratively testing and comparing variations of the design with low-fidelity prototypes. The idea of low-fidelity prototyping is that they are created without much detail and in most cases without any code being written, so that they can be created and modified very quickly.

For the Wikidata Query Builder, the initial design was tested as a paper prototype. Paper prototyping [Sny03] is a usability testing method that does not require a computer, and instead uses paper sketches of the user interface elements to quickly try out different page layouts, terms, and changes in the overall workflow. The sketches are typically done without much attention to details like straight lines, colors or icons and fonts.

The outcome of the short paper prototyping phase was a number of terminology decisions like "statement filter" and "qualifier filter", and the choice of the page layout. Generally, the layout was found to be most understandable the closer it was to that of Wikidata itself, with the exception of the nested statement filters, which posed the biggest source of confusion.

Since the paper version with its many small interchangeable elements quickly became unwieldy, a digital mockup version of the design was created using Balsamiq³. This provided an even quicker way of multiplying elements through copying and pasting, and adjustments of the overall design, however, moving elements around interactively was not quite as smooth as with the paper version. The final version can be seen in figure 5.4.

This mockup was used to test various scenarios in very short, focused test sessions. For these tests, the subject was shown a variation of the mockup with an incomplete query building process and given the task to take the next step,

³<https://balsamiq.com/>

5.2. Low-Fidelity Prototype

e.g. *Given this query with a single statement filter with property "instance of" and value "city", how do you think you could add another filter for specifying these city items to also have a "country": "Germany" statement?*

The image shows a browser window titled "Wikidata Query Builder" with the URL "http://query.wikidata.org/builder". The page has a header with navigation icons and a search bar. The main content area is titled "Wikidata Query Builder" and includes the subtitle "Create lists of Wikidata items by filtering them based on their statements." Below this, there are three main sections for building a query:

- Statement Filter:** A box containing a "Property" dropdown set to "instance of (subclass of)" and a "Value" dropdown set to "City". Below these is a "+ add qualifier filter" button.
- Statement Filter:** A box containing two "Property" dropdowns. The first is set to "head of government" and the second to "sex or gender". The "Value" dropdown for the first property is set to "any item matching...". Below the second property is a "Value" dropdown set to "female". There are "+ add qualifier filter" and "+ add statement filter" buttons below this section.
- Qualifiers:** A box containing a "Property" dropdown set to "end date" and a "Value" dropdown set to "does not exist". Below this is a "+ add qualifier filter" button.

At the bottom of the main content area, there is a "+ add statement filter" button and two buttons: "Show results" and "Show Query".

Figure 5.4: Mockup of a finished query generating a list of female mayors

Observations

The low-fidelity prototyping phase provided some important insights about the usability of the design concept. Adding statement filters, even multiple filters was usually not a problem for test subjects. Understanding the "any item matching..." special value and the change of context were clearly one of the bigger problems to be solved. One observation in this regard was that the connection of the indented statement became much clearer if it appeared directly as a consequence of the user choosing the "any item matching..." value. It was less well understood when users were presented with a finished query structure as the one in figure 5.4.

Another issue is the presentation of the different elements. Due to the amount of "add" buttons and text fields with different levels of indentations,

the structure quickly becomes confusing. An initial version of the prototype had no borders around the filters which was mentioned to make it difficult to tell which elements belong together. Another version with too many borders (additional borders around qualifier filters and nested statement filters), was described as "painful to look at". These challenges concerning the layout structure were considered during the creation of the high-fidelity prototype with the idea that the connections between elements can be conveyed better in a more detailed version of the design.

5.3 High-Fidelity Prototype

As opposed to low-fidelity prototypes, a high-fidelity prototype mostly focuses on details that are beyond the functionality and structural layout of the design such as colors, sizes, and fonts. When done on a computer screen, they also have the added advantage of providing a more realistic experience for interactions that are somewhat awkward on paper, like text fields that give suggestions based on the user's input. High-fidelity prototypes may include real functionality or a simple click-through version that creates the illusion of a working product [Gar10, p. 48].

The high-fidelity prototype for the Wikidata Query Builder was to be developed before the Wikidata Conference, which provided a great chance to gather qualitative feedback on the prototype. For this reason it was decided that the prototype should already contain a subset of its real functionality, to provide a more realistic experience and the opportunity to let people try out their own ideas. The features that were implemented were those that let the users explore the major parts of the user interface, including qualifier filters and nested statement filters with all properties that have the data type "item". The for now omitted features are literal data types and ordering of results, which were considered not to be necessary to evaluate the viability of the prototype.

The choices for fonts and colors were based on the Wikimedia style guide [wikd] to make the user interface look familiar to users, and to comply with the principles on good user interface design that were gathered by experienced designer working on Wikimedia software. The final design can be seen in figure 5.5, showing how the example from the introduction can be realized using the prototype.

The quantitative evaluation of the high-fidelity prototype was done at the Wikidata Conference and will be described in more detail in the Evaluation chapter (chapter 7). The initial round of qualitative feedback was gathered during the conference, and another major iteration with changes based on the feedback from participants of the conference was evaluated at the Wikimedia Deutschland office some weeks later.

5.3. High-Fidelity Prototype

The screenshot shows the Wikidata Query Builder interface in a browser window. The title is "Wikidata Query Builder" with the subtitle "Create lists of items by filtering them based on their statements". There is an "Examples" dropdown menu. The interface has two main filter sections. The first section has a "Property" dropdown set to "instance of" and a "Value" dropdown set to "Specific item" with the value "volcano". A blue tooltip is visible: "include subclasses (e.g. include instances of 'painting' when filtering for instance of 'work of art')". The second section has a "Property" dropdown set to "country" and a "Value" dropdown set to "Specific item" with the value "Japan". There are buttons for "+ add qualifier filter" and "+ add statement filter". Below the filters are sections for "Result columns" and "List options". At the bottom, there are buttons for "Show Results", "Show Query", and "Share Query". A table of results is displayed at the bottom.

Q wd:Q242730	Mount Asama
Q wd:Q733710	Mount Aso
Q wd:Q39231	Mount Fuji
Q wd:Q1067417	Mount Hakone

Figure 5.5: Wikidata Query Builder "List of volcanoes in Japan"

In both cases, test subjects were given multiple tasks, each of which had the goal to build a query to generate a list. Thinking aloud testing was used for these sessions, a usability test method that is considered to be one of the most valuable techniques for gathering qualitative feedback [Hol05]. By letting users speak out loud what goes through their mind while attempting to solve a problem with the application under test, this method does not only reveal *what* part of the design might lead to misconceptions during the process, but also *why* it was perceived this way by the subject.

The initial test of the high-fidelity prototype revealed some problem areas of the user interface that were not detected during the low-fidelity tests, likely due to the larger number of test subjects (18 during the conference), and better representation of users from the actual target audience. How these issues were

addressed will be described in the following section.

Feedback-based Improvements

The following three changes are based on reoccurring misconceptions that caused test subjects to temporarily get stuck during their thinking aloud tests. The cause for each of the problems was identified based on the actions and interpretation of their thoughts during the actions involving the problematic user interface elements. Different solutions were conceived and tested in another round of thinking aloud test sessions.

Special Values

In the initial version, special values for "item" properties like "any", "any item matching..." etc. were not in a separate field, and instead, as shown in figure 5.6, part of the item selector that would show the special values in its empty state when focused, and specific items after the user starts typing. Many test subjects did not know how to approach tasks that involved these special values, because they were only visible when the value field was focused, and because they did not notice them in previous tasks, where only specific item values were required.

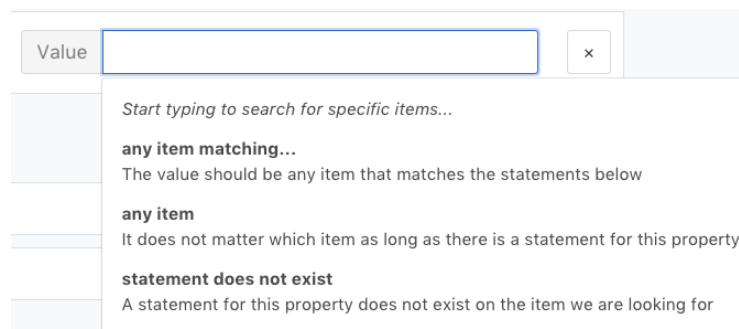


Figure 5.6: Special item values as part of the item selector

The solution for this issue can be seen in figure 5.5, where the special values got a separate select box and the item selector text field is only visible when "specific item" is selected. Since it is the most common case, "specific item" is selected by default. All testers that saw both variants agreed, that this was the more understandable solution.

In retrospect, this approach should have been obvious. In "User Centered System Design", Norman describes an approximation of the stages of user activity[ND86, p. 41], which states that users typically form their intention and specify their action sequence *before* they start executing the action itself. With the original solution, the option for the special cases is not visible in the application's initial state, and so the user is forced to "blindly" execute the first steps until after entering the property, when the value field is focused and

5.3. High-Fidelity Prototype

the special item value options are visible. This is also in line with Nielsen's heuristic "*recognition rather than recall*" [nie], which says that objects, actions and options should be visible, so that the user does not have to guess or remember how to take a certain action.

Special Properties

The issue with the presentation of special properties like "instance of any subclass of" or the recursive location property is similar to that of special values. The initial design included special properties in the suggestions, whenever the connected property was found, e.g. when the user typed "instance of" into the property selector, the suggestions would contain both "instance of" and "instance of any subclass of". Unfortunately, those users without prior SPARQL experience were mostly confused whenever these special properties appeared. The reason for this confusion is likely due to the fact that the problem, that these special properties solve, mainly occurs during the creation of SPARQL queries, and hence is not understood by the target audience.

Two different solutions were proposed to solve this issue:

- (a) Use the same design as for special values and show special properties in a separate field
- (b) Make the special versions of the properties the default for their related properties with an opt-out option (figure 5.7)

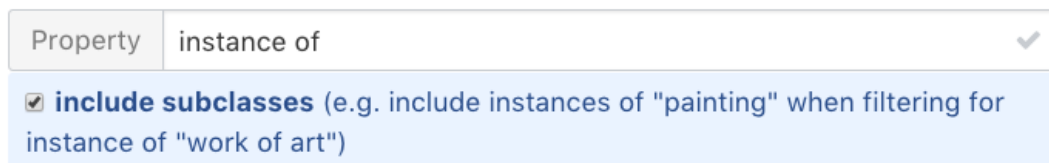


Figure 5.7: Solution to the special property problem – a hint describing the default option

The thinking aloud test of these two options showed, that option (a) did not actually solve the problem. Test subjects clicked these special properties when the task instructions were very suggestive that the simple property was not desired but otherwise ignored them. Another problem with this solution is, that special properties are very different from special values – while special properties are just variations of an existing properties, special values have their own functionality and are not in any way related to specific items.

The second solution, on the other hand, worked very well. Aside from being less confusing, test subjects also claimed that it helped them understand the problem it was solving. One might argue that this variation is another violation

of Nielsen’s *”recognition rather than recall”* heuristic, however, as mentioned above, the majority of the target users is not aware of this issue and therefore unlikely to look for an obvious solution.

Button Location

The final reoccurring problem was that multiple participants of the original test round accidentally clicked on the *”add qualifier filter”* button, when they actually wanted a statement filter. One or two of those even continued without immediately realizing their mistake.

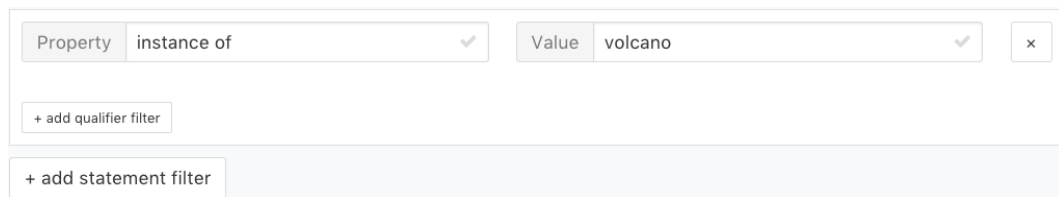
A screenshot of a web interface. At the top, there are two input fields: 'Property' with the value 'instance of' and a dropdown arrow, and 'Value' with the value 'volcano' and a dropdown arrow. To the right of the 'Value' field is a small square button with an 'x' icon. Below these fields, there are two buttons: '+ add qualifier filter' and '+ add statement filter'. The '+ add qualifier filter' button is positioned directly below the 'Property' and 'Value' fields, making it very close to the area where users were previously interacting.

Figure 5.8: Original location of the *”add qualifier filter”* button

The cause of the first problem is that the *”add qualifier filter”* button is closer to the property-value-pair that users previously interacted with. Tognazzini [tog] describes Fitts’s Law as *”the time to acquire a target is a function of the distance to and size of the target”*. The resulting usability guideline suggests that less frequently used elements should be smaller and further away from the user’s previous focus. Despite being a slightly smaller button, the *”add qualifier filter”* button is still much closer and in the same box as the property and value input field.

Again, two alternatives to this solution were tested:

- (a) A separate section for qualifier filters that is collapsed by default – the idea was to force users to read the headline before expanding the section (figure 5.9)
- (b) Move the *”add qualifier filter”* button to the side and make it even smaller (figure 5.10)

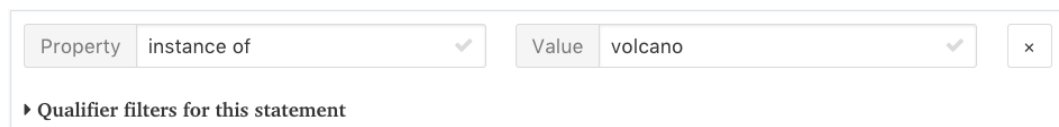
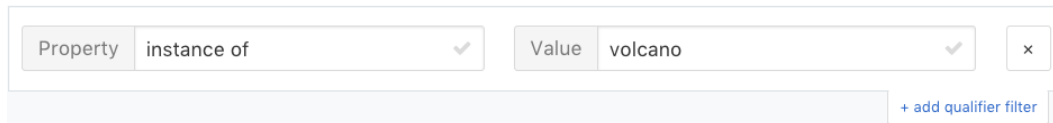
A screenshot of the same web interface as in Figure 5.8, but with a modification. Below the 'Property' and 'Value' input fields, there is a new section header: '► Qualifier filters for this statement'. This section is currently collapsed, indicated by the right-pointing triangle icon. The '+ add qualifier filter' and '+ add statement filter' buttons are no longer visible in this view.

Figure 5.9: Qualifier section collapsed by default

5.4. Summary



Property	instance of ✓	Value	volcano ✓	x
				+ add qualifier filter

Figure 5.10: Smaller qualifier button moved to the right side

Neither solution caused as much trouble in the second round of testing, however, the second solution was found to be superior, since it complied better to Fitts's law since solution (a) still shows the qualifier filters very prominently even though they are not expected to be used that much. In addition to moving the button, the size of the input fields was reduced in order to distinguish them better from those of the statement filters.

5.4 Summary

This chapter showed how the results of the analysis from the previous chapter were incorporated into the design. The description of the conceptual model showed how SPARQL features were mapped to Wikidata concepts in order to make them more understandable for the target users. Through low-fidelity prototypes, the ideas and concepts were built into a design that was iteratively tested and improved. A functioning high-fidelity prototype was developed and tested with real users at the Wikidata Conference, and the feedback used to make further improvements on the user interface.

6 Implementation

This chapter describes how the proof of concept prototype was developed. The first section provides a high-level model of the system architecture with its external interfaces, followed by a lower level description of the implementation of the web application, including the libraries and frameworks that are used.

6.1 System Architecture

When modeling the system architecture of web-based applications, one typically describes the involved internal and external services, databases, and communication between frontend and backend. Regarding the user interface as a single entity, the system architecture of the Query Builder is very simple. It is a single-page application¹ that does not need its own backend or storage. The frontend communicates with two external services: the Wikidata API and the Wikidata Query Service for getting the results of the generated SPARQL queries (figure 6.1).

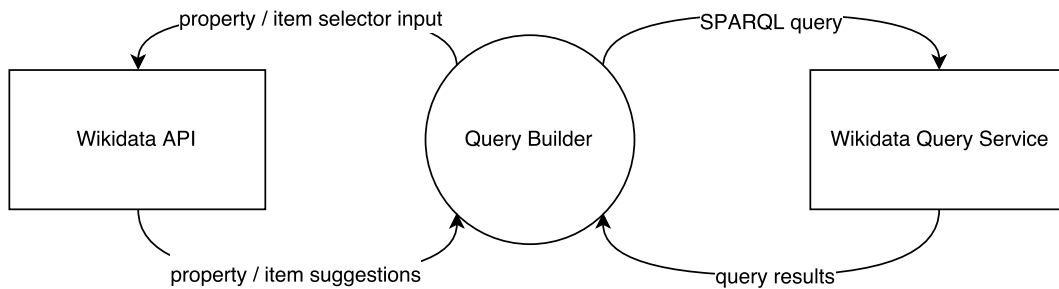


Figure 6.1: Data flow between external services and the Query Builder

The Wikidata API is used for suggesting properties and items to the user based on the input in the property selector and item selector. Using the `wbsearchentities`² API action, the search term from the input field and setting the `type` parameter to either "item" or "property" respectively. Due to the backend-less architecture of the application JSONP is used, which enables read-only cross-site requests to the MediaWiki API [cro] and allows the response to be processed directly within the frontend of the Query Builder.

It was considered to also directly access the SPARQL endpoint to retrieve a JSON serialization of the results for the produced queries, however, since

¹https://en.wikipedia.org/wiki/Single-page_application

²<https://www.wikidata.org/w/api.php?action=help&modules=wbsearchentities>

6.2. Technical Implementation

the Wikidata Query Service provides a convenient way to produce tables from queries that can be encoded as a URL parameter, this was not necessary. This HTTP endpoint `https://query.wikidata.org/embed.html#QUERY`, where `QUERY` was replaced with a URL encoded version of the generated query, was embedded into the Query Builder website using an `iframe`-tag that is re-rendered whenever the user clicks the "Show results" button.

6.2 Technical Implementation

Contrary to the straightforward overview of the involved services, the user interface involves quite a lot of interfacing entities. It is entirely written in JavaScript, using the Vue.js³ framework, which provides tools for highly interactive, component-based user interfaces. Similar to the React⁴ framework, it supports a declarative style for the creation of component templates, which are synchronized automatically with their corresponding data, when rendered in the browser.

6.2.1 User Interface Components

Components, in this context, refer to a concept that is related to Web Components [web] – custom HTML elements that have their own behavior and state. Like normal HTML tags, Vue components can be integrated into other templates using the syntax that is familiar from HTML. An example of this can be seen in the excerpt from the Query Builder's `StatementFilter` component template in listing 6.1.

```
1 <div class="column">
2   <PropertySelector v-on:select="selectProperty"
3     :initial="statement.getProperty()"
4     ref="property">
5   </PropertySelector>
6 </div>
7 <div class="column">
8   <ValueSelector v-on:select="selectValue"
9     :disabled="valueSelectorDisabled"
10    :objectId="statement.getId()"
11    :initial="statement.getValue()"
12    ref="value">
13   </ValueSelector>
14 </div>
```

Listing 6.1: Excerpt of a template of the `StatementFilter` Vue component containing multiple child components

³<https://vuejs.org/>

⁴<https://reactjs.org/>

This example shows how the `PropertySelector` and `ValueSelector` component are embedded in the `StatementFilter` component. The attributes passed to the components follow a similar syntax as HTML attributes and provide a way to pass data from parent to child component and register event listeners. Structuring the user-facing parts of the application as Vue components allows for a natural modularization of code and reusability. The same components for property selector and value selector are reused in the component for the qualifier filters.

Using the `vue-devtools`⁵ browser extension, the structure and state of all components can be inspected and manipulated for debugging purposes. Figure 6.2 shows the structure of the component hierarchy of the Query Builder during the creation of a query with 3 statement filters that generates the "List of female mayors" (list of items with "instance of": "city", "head of government": any item matching "sex or gender": female"). Note how `Statement` (statement filter) and `StatementList` (list of statement filters) form a recursive data structure, where a statement filter can contain a list of statement filters whenever the "any item matching..." special value is selected in the parent statement filter's value selector.

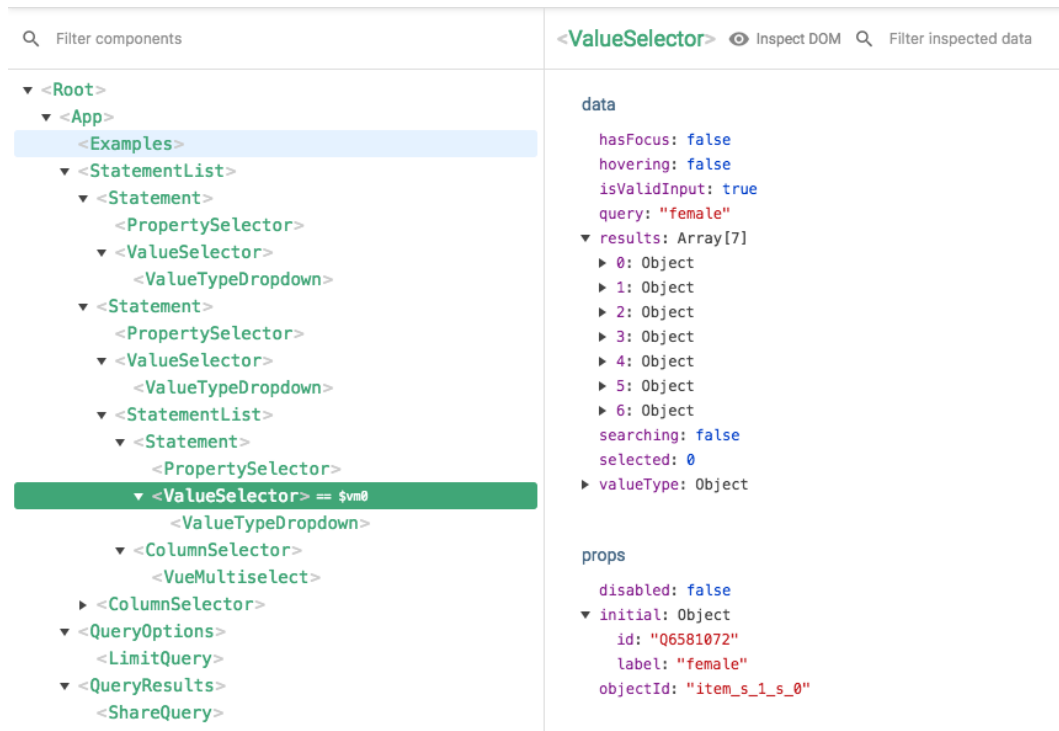


Figure 6.2: Query Builder component hierarchy during the creation of a complex query

⁵<https://github.com/vuejs/vue-devtools>

6.2.2 State Management

The application stores the state in two different ways – either globally or locally to a component. State that is stored in the local context of a single component contains data that is relevant only to the component itself, e.g. in the case of a property selector it may reflect whether it is currently focused, what properties have been suggested for the user input and the like. Global state is shared between components and in the Query Builder’s case contains all the information necessary to build the SPARQL query.

Global state management follows the Vuex pattern [vue], which provides a framework for a centralized store with predictable state mutations. The Vuex pattern encourages a one-way data flow from the *view*, to *actions*, to the *state* and back to the *view*, as seen in figure 6.3. The view calls the store’s actions, which update the store state, which in turn is reflected in the view. This has the effect that components never directly manipulate the state of the central store, and instead interface with it through the store’s *actions*. Since the store is central and multiple components may have common state, one action may affect multiple views.

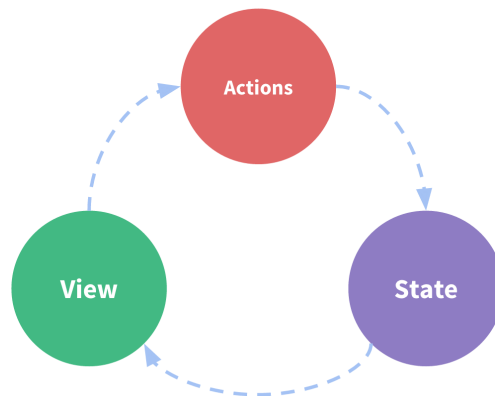


Figure 6.3: Simplified Vuex data flow

For the Query Builder, following this pattern had two major benefits:

1. Communication between components happens through state manipulation, and complicated event handling across components is avoided. This is especially evident when generating the SPARQL query by clicking the “Show Results” button: Since all the required information is written to the central state during the query building process, the component generating the query does not have to collect it from all the other components, and can simply access it through the central store.
2. Due to the circular flow between view and store, not only does the view create new state, but also the other way around. The Query Builder

has a feature for sharing queries, which enables the export and import of query builder state. When clicking the "Share Query" button, the store's state is serialized into a JSON structure, which is appended to a parameter of the Query Builder's URL. From this URL parameter, the Query Builder can deserialize and initialize its entire state and present the user who followed a share link with the same set of statement filters and input that the original sharer created.

6.2.3 SPARQL Query Generation

Aside from the interactive user interface and its import and export functionality, generating the actual SPARQL query was the most intricate part of the implementation of the Query Builder. Every possible combination of filters and property-value pairs needs to produce a semantically equivalent SPARQL query. For this proof of concept prototype version, readability of the query and performance optimization were not treated as primary concerns.

Query Builder State

As mentioned in the previous section, the query was built from the central state that is created during the query building process. The application relevant application state is represented in the form that can be seen in listing 6.2 in JSON notation.

```
1  {
2    statementFilters: {
3      subjectId: {
4        statementFilterId: {
5          property: PropertyObject,
6          value: ValueObject
7        },
8        // ...
9      },
10     // ...
11   },
12
13   qualifierFilters: {
14     statementSubjectId: {
15       qualifierFilterId: {
16         property: PropertyObject,
17         value: ValueObject
18       },
19       // ...
20     },
21     // ...
22   }
23 }
```

6.2. Technical Implementation

Listing 6.2: Data representation of application state relevant for query generation

The state contains one map object for statement filters and another one for qualifier filters – both following the same structure. The `statementFilters` object maps subject IDs to maps, that have statement filter IDs for keys and contain a property-value pair as their value. The comment in line 9 indicates that there may be multiple statement filters mapped to each subject and the one in line 11 suggests that there may be multiple subjects. New subject maps are added to the state whenever the special value "any item matching..." is chosen for any of the values, thereby creating a new path pattern, that connects an object variable to a new subject variable. A new statement filter is added to the subject map, whenever the user clicks the "add statement filter" button.

Note that despite the query builder user interface following a nested structure for statement filters, the depth of the state always remains the same. The nesting is instead reflected in the state through the parent filter's value ID being identical to the child's subject ID. This has the benefit that it is closer to the way SPARQL triple patterns are constructed.

`PropertyObject` and `ValueObject` are polymorphic objects. Their class depends on the type of property and value that were chosen in the query builder. Properties are either special (Wikidata property + property path) or normal Wikidata properties. Value objects are either special ("any", "any item matching...", "not exist..."), item values or literals (string, date, integer...).

Query Generation

Since it is considered the most intricate part, only the generation of the basic graph pattern is generated from statement and qualifier filters is described in this section, omitting the possible addition of `FILTER`, `VALUES`, `OPTIONAL` and other constructs.

The content of the `addStatementFilterTriples` method of the `QueryGenerator` class is described in the following listing

```
1 for (const subject in this.statementFilters) {
2   for (const statementFilterId in this.statementFilters[subject]) {
3     const statementFilter = this.statementFilters[subject][
      statementFilterId]
4
5     if (statementFilter.getProperty() && statementFilter.getValue()) {
6       const statementFilterVariable = this.makeStatementFilterVariable(
        statementFilterId)
7
8       query.addTriple(
9         `?${subject}`,
```

```

10     statementFilter.getProperty().getStatementPredicate(),
11     statementFilterVariable
12 )
13
14 query.addTriple(
15     statementFilterVariable,
16     statementFilter.getProperty().getStatementValuePredicate(),
17     statementFilter.getValue().getObject()
18 )
19 }
20 }
21 }

```

`this.statementFilters` contains the statement filter state as described in the previous section. The outer loop iterates over all subjects, whereas the inner loop iterates over each statement filter of the subject, creating a `statementFilter` variable in line 3. Statement filter IDs are unique within the query, so that they can be used as variables within the query. Line 6 shows how the helper method `makeStatementFilterVariable` creates a variable from said ID by simply prepending a "?" and appending "Statement".

In the next step, two triple patterns are added: one triple pattern for connecting the subject to the statement, and another triple pattern connecting the statement to its value. Note that different methods are called on the property object for each triple pattern in order to produce a predicate with the correct namespace prefix. Whether this predicate contains a property path construct is abstracted through the polymorphic property object.

In the second triple pattern, `getObject()` is called on the value of the statement filter. Depending on the type of value, this produces either a variable, a Wikidata item id with the "wd:" prefix, or a literal.

This method produces the query shown in listing 6.3 from the statement filters shown in figure 6.4. This example includes property paths, two statement filters with the same subject, and one nested within another.

The screenshot shows a Query Builder interface with three filters. The first filter has a Property of 'instance of' and a Value of 'Specific item' with a qualifier 'city'. Below it, there is a checkbox for 'include subclasses' with a tooltip that says '(e.g. include instances of "painting" when filtering for instance of "work of art")'. The second filter has a Property of 'head of government' and a Value of 'any item matching...'. The third filter is nested under the second, with a Property of 'sex or gender' and a Value of 'Specific item' with a qualifier 'female'. Each filter has an 'add qualifier filter' button.

Figure 6.4: Query Builder user interface state that produces the query in listing 6.3

6.2. Technical Implementation

```
1 SELECT DISTINCT ?item ?itemLabel ?item_s_1 ?item_s_1Label
2 WHERE {
3     ?item p:P31/wdt:P279* ?item_s_0Statement .
4     ?item_s_0Statement ps:P31/wdt:P279* wd:Q515 .
5
6     ?item p:P6 ?item_s_1Statement .
7     ?item_s_1Statement ps:P6 ?item_s_1 .
8
9     ?item_s_1 p:P21 ?item_s_1_s_0Statement .
10    ?item_s_1_s_0Statement ps:P21 wd:Q6581072 .
11
12    SERVICE wikibase:label { bd:serviceParam wikibase:language "en" . }
13 }
```

Listing 6.3: Query generated from the query builder shown in figure 6.4

This example also shows that every query uses the `DISTINCT` modifier to avoid duplicate list entries by default, and includes the label `SERVICE` construct to provide an English label for every item in the result set. These two additions are always added by default and cannot be modified in the current version of the prototype. In future versions, it might be desirable to make the language and language fallbacks configurable.

It should be noted that each of the three triple pattern pairs in listing 6.3 can also be expressed as a single pattern by removing the statement variable and using the `wdt:` prefix, e.g. `?item p:P123 ?statement . ?statement ps:P123 wd:Q234` becomes `?item wdt:P123 wd:Q234`. This was not done, because always having the statement variable available facilitates the addition of qualifiers. Adding a conditional checking for existing qualifiers would indeed improve readability and performance of the query, but those criteria were not a primary concern, as previously mentioned.

Listing 6.4 shows how triple patterns are generated from the `qualifierFilters` object of the Query Builder's internal state. This method work similarly to the previously shown one, except that it only adds a single triple pattern for each qualifier filter, generates the `makeStatementFilter` helper method to produce the variable for its subject and calls `getQualifierPredicate` on the property to produce a predicate with the `pq:` prefix.

```
1 for (const subject in this.qualifierFilters) {
2   for (const qualifierId in this.qualifierFilters[subject]) {
3     const qualifierFilter = this.qualifierFilters[subject][qualifierId]
4     const statement = this.makeStatementFilterVariable(subject)
5
6     query.addTriple(
7       statement,
8       qualifierFilter.getProperty().getQualifierPredicate(),
9       qualifierFilter.getValue().getObject()
10    )
11  }
```

Listing 6.4: Generating triple patterns from qualifier filters

6.3 Summary

With the application being entirely backend-less and interacting with only two external services through very simple interfaces, the implementation of the Wikidata Query Builder mostly revolves around the interactive query building user interface and the creation of the SPARQL query from the user input.

The user interface was built using modern JavaScript frameworks for the application's state management and interactive elements. A component-based structure enables modular code organization and reusable modules, that interact mainly through a central store, following the Vuex pattern.

The section on query generation shows how the application stores statement filters and qualifier filters, which provide the data for producing the query. The algorithm that turns these data structures into triple patterns and finally into a SPARQL query is explained based on an example. The code is published on GitHub⁶ under GNU General Public License v2.0.

⁶<https://github.com/jakobw/wd-query-builder>

6.3. Summary

7 Evaluation

While the goal of *formative* evaluation is to iteratively improve and reassess a product's usability during its design phase [LFH17], *summative* evaluation is done as a final assessment after all major design decisions have been made. Summative evaluation is typically done either as a comparative study, testing several products against one another, or tested in a standardized way, so that the result is comparable to similar studies [sum], aiming to reveal whether or not the final solution is useful and usable.

One established way of getting a comparable, user-driven rating is by using standard questionnaires such as the System Usability Scale (SUS) [B⁺96], the Usability Metric for User Experience (UMUX) [Fin10], or the User Experience Questionnaire [LHS08]. Among the three, SUS is the most popular choice of questionnaire for such evaluations, however, it was considered too generic for the application in this scenario. Like SUS, UMUX focuses on the ISO 9241 definition of usability based around effectiveness, efficiency and satisfaction, and with only 4 questionnaire items yields even more generic and harder to interpret results. For this project's quantitative evaluation, the User Experience Questionnaire was chosen, which is arguably more modern and more detailed than UMUX and SUS.

7.1 User Experience Questionnaire

UEQ consists of 26 questionnaire items, each of which belongs to one of the six factors that represent the authors' definition of user experience [LHS08]. Out of those factors, three belong to the category of pragmatic qualities (*efficiency*, *perspicuity*, *dependability*), two belong to the category of hedonic qualities (*stimulation*, *novelty*), and *attractiveness* as an unattached factor (figure 7.1).

- *Attractiveness* is supposed to describe the user's overall impression of the product
- *Perspicuity* tells whether or not the user was easily able to use the product
- *Efficiency* rates the effort that it took to complete the tasks
- *Dependability* describes whether the user felt in control of the interactions with the application
- *Stimulation* answers the question of whether the program was exciting and motivating to use

7.2. Usability Test Setup

- *Novelty* expresses the user's judgment of the product's innovativeness and creativity

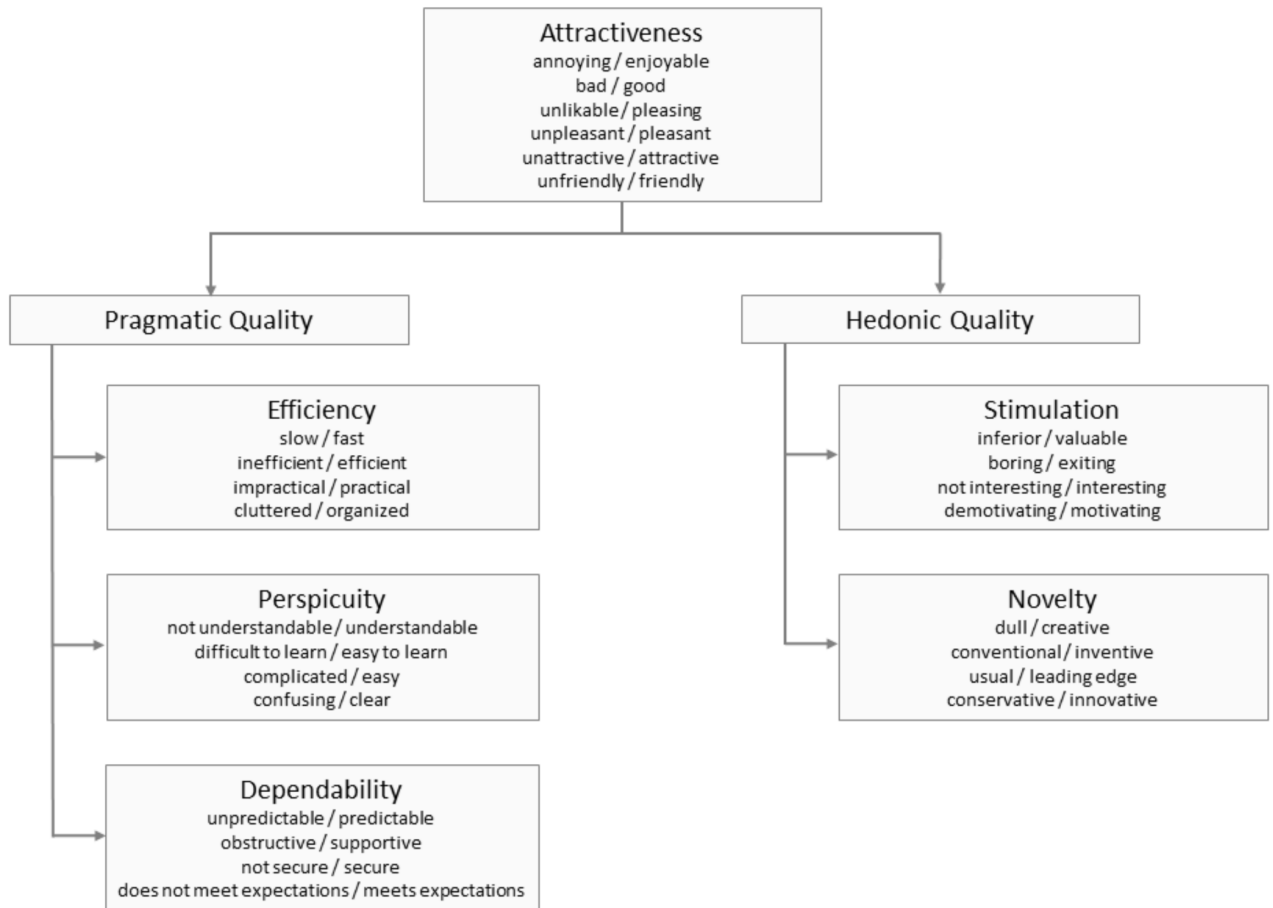


Figure 7.1: UEQ categories, scales, and items (picture from [Sch15])

Every item on the UEQ consists of a pair of adjectives, one that describes a characteristic of user experience, and its antonym. The pairs are then rated on a seven stage scale.

7.2 Usability Test Setup

The usability test took place at the WikidataCon¹, the first Wikidata conference with over 200 attendees from the Wikidata community, which presented a good opportunity to gather feedback from many potential users. A small table was reserved for 6 hours from 12:00 - 18:00 on the first day of the conference inside a relatively quiet room. Materials used were a laptop with a 15" screen,

¹https://www.wikidata.org/wiki/Wikidata:WikidataCon_2017

printed out task descriptions and questionnaires, and an envelope to store the filled out questionnaires.

The usability tests were conducted with individual people and were expected to take about 15 to 20 minutes per person. Due to a busy schedule and logistical challenges, individual scheduling was not possible, and so test subjects had to be processed according to "first come, first served" principle.

Each individual test went through four steps:

1. Introduction
2. Thinking Aloud Test with 3 Tasks
3. Debriefing
4. UEQ

The introduction was there to inform each participant about the main idea behind the Query Builder, and the context and procedure of this test. Before the actual test started, the user was asked to speak their thoughts on the product and their intentions out loud while they solved the task. After all three tasks were completed, there was a short debriefing, thanking the participant for their interest and participation and asking them to fill out the questionnaire and to return it to the provided envelope where the results were collected. Where to fill out the questionnaire was left to the participant to keep the answers anonymous and unbiased.

Tasks

The three tasks of increasing level of difficulty each were about generating a list from start (empty query builder) to finish (showing results). They contained a short description of the task, an alternative description containing the labels of the Wikidata property and item names that were supposed to be used, and a screenshot of an example item that would be an element of the generated list showing statements that would be relevant during the query building process.

The alternative description was added so that people would not waste time first trying to find out how the relevant piece of knowledge was modeled in Wikidata, since that was not helpful for the usability evaluation of the Query Builder. The screenshots were added for a similar reason – from personal experience and consulting experts it was found that one typically builds queries by looking at an example item that is similar to those that would make up the list elements and then pick out the statements that describe it as such. In the following, the tasks and their descriptions are presented. The screenshots of the example items for each task can be found in the appendix.

7.3. Results

Task 1: Cats

The first task required only a single statement filter. It was mainly there to get the test subjects acquainted with the user interface and see if the general structure is understood and where to enter the property and value, and how to show the list of results.

Task description: Create a list of cats

Alternative description: Build a query to find items that have an "instance of" statement with "cat" for its value.

Task 2: Volcanoes

The second task was more complex because it required the use of two statement filters and thereby a certain level of understanding how the filters worked, and finding the "add statement filter button".

Task description: Create a list of Icelandic volcanoes

Alternative description: Build a query to find items that have an "instance of" statement with "volcano" for its value, and a "country" statement that has "Iceland" for its value.

Task 3: Female mayors

The last task was the most difficult one. It required the use of the "any item matching..." special value for the head of government statement filter and understanding the context switch when describing value items with indented statement filters.

Task description: Create a list of cities that have female mayors

Alternative description: Build a query to find items that have an "instance of" statement with "city" for its value and a "head of government" statement, which has an item value that has a "sex or gender" statement with "female" for its value.

7.3 Results

The results of the qualitative feedback of the thinking aloud usability test have already been discussed in section 5.3, so this section only focuses on the results around the task completion and the analysis of the questionnaire results. During the 6 hours time slot, 18 conference attendees participated in the usability test, 16 of which were willing to fill out the user experience questionnaire.

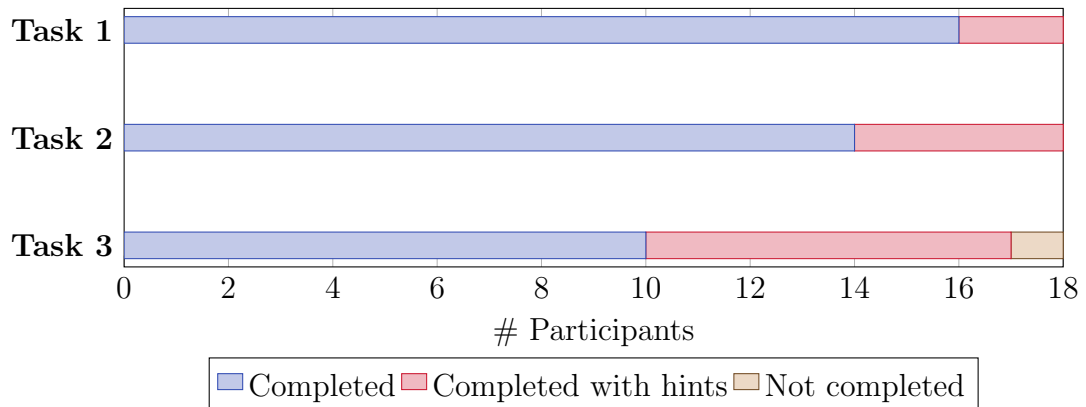


Figure 7.2: Distribution of participants' task completion status per task

7.3.1 Task Completion

As figure 7.3.1 shows, the majority of participants was able to finish the presented tasks. The first task posed no problem to most people. Confusion about the instructions was not considered giving hints, e.g. there was a participant who refused to read the alternative description in order to challenge himself and thought a list of classes of cats (domestic cat, lion, tiger...) was requested.

The people that needed hints for the second task were mainly those that clicked the "add qualifier filter" button instead of the one for statement filters and did not immediately realize it on their own. The hints that were given for the "completed with hints" section were not concrete instructions for what was to be done, but instead nudges in the right direction like *"Did you intend to add this property-value pair as a qualifier?"*.

As expected, task 3 and especially the "any item matching..." posed the biggest obstacle. This test was conducted before the improvements discussed in section 5.3, and most test subjects struggled with the identification of the option of the special item value, and hesitated before even starting to build a solution. The hint that was sufficient for most test subjects that got stuck in this situation was *"Try filling out the query builder as far as it makes sense to you."* and typically, after confirming the "head of government" property, they paused and saw the options for special values with their respective descriptions. Only one test subject got stuck entirely and demanded to reveal the solution.

7.3.2 UEQ Results

The User Experience Questionnaire provides a data analysis tool in the form of an Excel document that facilitates the evaluation. It requires only the results from each of the handed out questionnaires as input and generates visualizations, and computed values for mean scores, confidence intervals, and comparison to other results from the UEQ benchmark.

7.3. Results

The rating for each questionnaire item on the seven stage scale is converted to values in the range of -3 (most negative) to +3 (most positive). The overall results for each scale can be seen in figure 7.3. The bars show the mean value, alongside the confidence interval at a 95% confidence level. As the chart shows, all the measured mean values except that of novelty are considered positive.

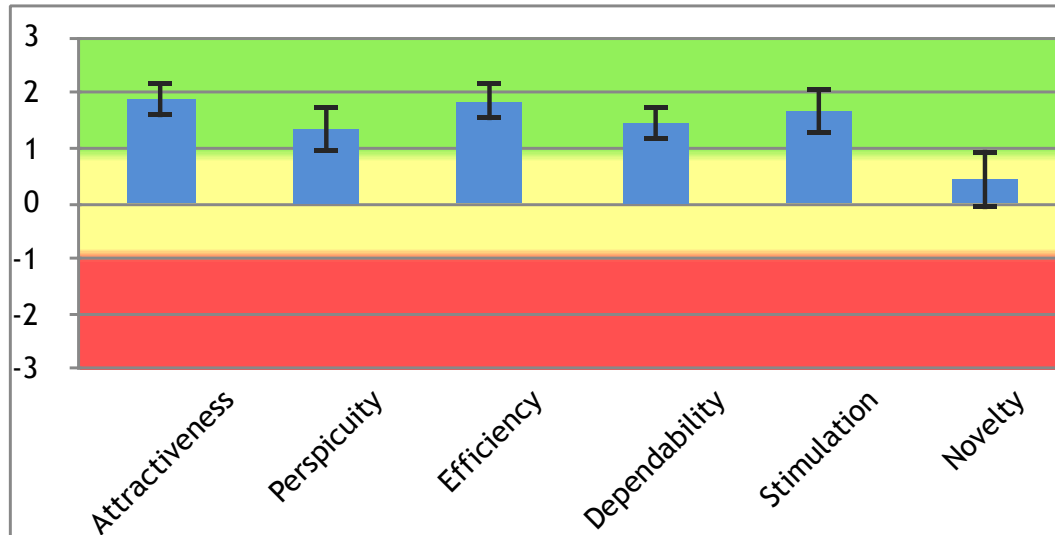


Figure 7.3: UEQ results for the 6 scales, visualized by the provided data analysis tool

An interpretation of the mean values compared to the benchmark that is also provided by the UEQ data analysis tool can be seen in table 7.3.2. The benchmark contains the data of 246 product evaluations with the UEQ with a total of 9905 participants in all evaluations [Sch15]. According to this comparison, the Query Builder achieved *excellent* results (in the range of 10% best results) for attractiveness, efficiency and stimulation. For perspicuity and dependability the scores are considered *above average*, meaning that they are better than the mean values of 50%, but worse than 25%, and novelty got a *below average* interpretation (at least 50% are better and at least 25% are worse).

It should be noted that the main goal was to achieve good results in the pragmatic qualities (efficiency, perspicuity, dependability), and hedonic quality and attractiveness were not of primary concern. Since all three scales belonging to pragmatic quality got above average results, the current design approach is considered successful. The excellent ratings for attractiveness and stimulation are an added bonus, and the low score for novelty is unfortunate, however, it is not an issue that needs to be addressed immediately.

Table 7.3.2 shows that the standard deviation and confidence values for perspicuity, stimulation, and novelty are quite high. It is likely that test subjects that struggled with the last task rated the questionnaire as less understandable than those who did, which leads to a higher deviation for perspicuity. It is

Scale	Mean	Rating
Attractiveness	1.875	Excellent
Perspicuity	1.344	Above Average
Efficiency	1.844	Excellent
Dependability	1.438	Above Average
Stimulation	1.672	Excellent
Novelty	0.438	Below Average

Table 7.1: Questionnaire results per scale with comparison to results from the UEQ benchmark

unclear if any particular part of the setup or the application itself lead to the low agreement on scores for stimulation or novelty, or whether the population size of 16 was not sufficient. In the UEQ Handbook [Sch15], the recommended number of participants for such a study is 20 - 30 persons, which unfortunately was not possible due to time constraints.

Scale	Mean	Std. Dev.	Confidence	Conf. Interval	
Attractiveness	1.875	0.651	0.319	1.556	2.194
Perspicuity	1.344	0.903	0.443	0.901	1.786
Efficiency	1.844	0.706	0.346	1.498	2.190
Dependability	1.438	0.680	0.333	1.104	1.771
Stimulation	1.672	0.907	0.444	1.227	2.116
Novelty	0.438	1.094	0.536	-0.098	0.973

Table 7.2: Standard deviation and confidence intervals ($p = 0.05$) per scale

The general impression was that people who got to test the application and fit in the target audience were excited about the product and found it useful. Altogether, based on the verbal feedback that was received and inquiries about additional features, the Query Builder was very well received.

7.4 Summary

Through extensive research and the application of user-centered design methods in the design and development phases, the Query Builder was built to be a product that is usable and useful from the start. The formal evaluation at the WikidataCon was carried out to learn whether the usability standards had been achieved.

Over a 6 hour time period, 18 thinking aloud tests have been carried out, each of which required test subjects to complete 3 tasks ranging from building a simple query with one statement filter, to a complex nested statement filter structure. Most participants were able to complete the tasks and gave valuable qualitative feedback.

7.4. Summary

In order to gather quantitative data, the User Experience Questionnaire (UEQ) was chosen over the more generic System Usability Scale and the Usability Metric for User Experience due to a higher level of detail. Out of the 18 test participants, 16 filled out the questionnaire. In 3 out of the 6 categories which make up UEQ's definition of User Experience, the Query Builder was rated *excellent*, 2 categories got *above average* scores, and one *below average*.

8 Discussion

The response of the sample of people from the Wikidata community and the quantitative feedback from the questionnaire showed promising results. This chapter discusses possible improvements to the overall research approach and to what extent the research goals have been achieved.

8.1 Evaluation of this Work

The goal of this thesis was to build a graphical user interface that allows experienced Wikidata users to build SPARQL queries, to generate lists for Wikipedia articles. Another goal was to give the reader an overview of existing SPARQL query builder projects and provide a categorization based on their approaches.

Studying the related projects revealed that none of the other projects fit the exact requirements of the Wikidata Query Builder and were either too simple and unable to produce the queries with features that are commonly used in the analyzed data set, or only suited for advanced users with extensive knowledge of the SPARQL language. Especially noticeable was the lack of projects that provided abstractions on top for a certain RDF schema to simplify the query building process. This may be an indication that flexibility was higher prioritized than ease of use for these projects. Nevertheless, exploring the solution to various design problems and different interface paradigms provided valuable insights about possible trade-offs between simplicity and functionality that needed to be considered in the design phase.

Analyzing the SPARQL queries from Listeria shed light on user needs, that were not previously expected. Before the analysis phase, it was speculated that simple star-shaped query patterns may be sufficient to generate most of the queries, however, the analysis revealed that paths (subject-object joins) frequently appear. This is analogous to the findings of Hernández et al. [HHR⁺16], that most queries from the Wikidata Query Service examples form tree shapes.

Due to time constraints, it was not considered that the queries from the Listeria Bot analysis might differ in some cases from queries that are needed for generating lists for Wikipedia articles. Since the current approach of the Listeria Bot was not accepted by the community [lisa], it is currently suspended from some Wikipedias' article name space, and only used experimentally on project pages, and on user pages. As a consequence, it is hard to tell without further analysis how many of the lists are similar to those found on Wikipedia articles, and how many are created by editors to monitor and manage their

8.2. Research Contribution

maintenance work.

The results of the quantitative UEQ data were above average for all categories that were considered important. It is unclear, whether the results may have been influenced by the generally pleasant atmosphere at the WikidataCon or the sample of people that have participated. Unfortunately, a survey for demographic data was not carried out in combination with the questionnaire at the conference, so that no correlations between skill level or other attributes and the respective results can be observed.

All things considered, the goal of creating a working prototype for a query builder for list generation following a user-centered design approach has been successful. The resonance during testing sessions, especially with less experienced subjects, was positive and there were many inquiries, when a final version would be released. Despite the lack of an official feature for generating lists from SPARQL in Wikipedia, the tool has already proven useful for building experimental queries and exploring the data.

8.2 Research Contribution

The main research contribution of this work lies in the assessment of state-of-the-art query building tools and their categorization. 13 different query builders have been analyzed and classified by their user interface paradigm. Out of the 13, those that provided an operating version were examined further based on specified user interface attributes.

The hierarchical categorization, based on that of Haag et al. [HLBE14], can be reused and extended in future research. The distinction between *explorative* and *triple pattern based* query builders, further differentiated by the structural representation of the query's graph pattern has proven a sensible systematization. Additionally, the more detailed evaluation of tools based on the feature, interface, and interaction attributes has provided a basis for a framework for assessing query builder tools.

A novel mapping from Wikidata concepts to SPARQL query constructs has been created in the process of designing the user interface for the Wikidata Query Builder. Through statement filters, Wikidata users can build queries using concepts they know from Wikidata and everyday web browsing experience. This approach of simplifying the query building process by using abstractions specific to the RDF schema of the SPARQL endpoint, and data model of knowledge base is an approach that was rarely found during the inspection of related tools and may be used to inspire similar mappings.

Finally, this advancement in the realization of automated list generation on Wikipedia can spark further discussion to find a solution for the technical and practical concerns that emerged in the discussions around Listeria.

9 Conclusion

This thesis presented the creation of a graphical user interface for creating SPARQL queries for automated list generation using Wikidata’s data. The following sections summarize the work that has been done, give an overview of its limitations, and an outlook on possible future work.

9.1 Summary

Chapter 2 gives an account of the necessary theoretical foundation required for the following chapters. RDF and SPARQL were described as the central technologies in this work – RDF being the standard for representing data on the linked web, and SPARQL the most commonly used query language for it. For the analysis and design part, it was also important to give an overview of Wikidata’s data model, and how it is represented in RDF.

The related literature was reviewed in chapter 3. It shows how data from Wikidata is incorporated in Wikipedia and what similar work has been done in the past. For background information on SPARQL analysis, several works with varying approaches and different data sources are presented, focusing on feature-based and pattern-based analysis. Finally, an overview of other query builder tools is presented, concluding with the proposed categorization from Haag et al. [HLBE14].

Based on the theoretical foundations and the previous work from the related literature, chapter 4 described the analysis part, where the data from the Listeria SPARQL queries were examined in order to find out how SPARQL is currently used for experimental list generation, and which particular features and graph patterns need to be supported. The results from this analysis, combined with input from Wikidata’s product management, make up the requirements for the Query Builder. Using the categorization from the related literature as a reference, an alternative categorization into experimental and triple pattern based query builders is proposed. Using the requirements as a reference, a selection of these similar tools is assessed in order to determine to what extent certain approaches fulfill the requirements for the Wikidata Query Builder.

After learning the requirements and the strengths and weaknesses of certain user interface paradigms for query builders, the creation of the conceptual model was demonstrated in chapter 5. A novel mapping from Wikidata concepts to SPARQL query constructs was used to facilitate the query creation for users familiar with Wikidata’s data model. A low-fidelity prototype was created to test the fundamental ideas of the design, and was further enhanced

9.2. Limitations

by a high-fidelity prototype, that was tested in a user study at the Wikidata-Con using thinking aloud testing. Both prototyping phases generated valuable input on the design, and respective improvements have been tested and implemented.

Chapter 6 – "Implementation" gives the reader insights into the software behind the Query Builder. Modern JavaScript frameworks were used for the interactive user interface and the application's state management. Besides the description of the user interface, a code example explains how the resulting SPARQL query is generated from the Query Builder's internal state.

The following chapter 7 outlines the evaluation of the high-fidelity prototype at the Wikidata conference. 18 test subjects participated in the thinking aloud test, 16 of which also filled out the questionnaire which was used to gather quantitative data to evaluate the prototype's usability. The standardized User Experience Questionnaire was used as it provided a good compromise between detail and conciseness, and also provided a benchmark for comparing the results to previously evaluated products. The evaluation showed that it scored above average according to the benchmark in all areas, that were considered important. These encouraging results and the generally positive feedback from testers suggest that the prototype meets the community's needs and that the design efforts were successful.

9.2 Limitations

As for limitations, it should be noted that the work of this thesis is limited to the creation of SPARQL queries, and does not include the automatic generation of the list itself in the context of Wikipedia or MediaWiki. The list can be seen in the results of the query builder, or the Listeria Bot in combination with the Wikidata list template can be used to embed the list into a Wikipedia page.

The application that was implemented as part of this work was deliberately framed as a prototype, and not a production-ready web application. The reason behind this was to focus on the fundamental concepts of the design and quickly getting a working proof of concept, rather than producing a fully feature-complete product, which would require a considerably higher planning and implementation effort.

Additionally, the scope of the survey was limited to gathering quantitative feedback from the UEQ. Usually, such a survey also involves the collection of demographical data, which unfortunately was not done at the conference, and could not be repeated due to the lack of an equivalent opportunity within the given time frame. Another evaluation would also improve the confidence of the measured usability scores.

9.3 Future Work

Future work based on the results of this thesis can follow on two parts — the categorization and assessment of other query builder projects, and the continued implementation and evaluation of the Wikidata Query Builder. The categorization can be further extended to accommodate projects that are fundamentally different from the ones analyzed in chapter 4. Similarly, the defined attributes can be reused to build a more detailed framework for the evaluation of query builder interfaces.

The positive results from the evaluation that took place at the WikidataCon can be further supported by performing another user study, ideally with more than 20 participants. The evaluation done in this work was mainly focused on the questionnaire results, and could be improved by a comparative summative evaluation, that compares the Wikidata Query Builder to similar query builder tools to see whether it performs better than those in common scenarios. Since test subjects that fit the target audience are rarely in one place in larger numbers, a Wikimedia conference such as the Wikimania¹ provide a good opportunity to carry out further tests.

As mentioned in the discussion chapter, the data used in the SPARQL query analysis likely contains a subset of queries that are used for maintenance work by Wikipedia editors, rather than the generation of lists as they appear within Wikipedia articles. In future work these queries could be filtered out, or tested to be sure of the ratio of maintenance lists to lists as they would appear on Wikipedia’s article name space.

A random sample of Wikipedia lists could be used to further validate that the Query Builder is indeed capable of building the majority of lists. At present, this has only been tested through expert consultation and the *Listeria* query analysis.

Finally, the implementation of the Query Builder has to be finished and released. Only then it will show whether the tool is truly useful to the community, and whether it can advance Wikimedia’s vision of improving both Wikipedia and Wikidata through automated list generation.

¹<https://en.wikipedia.org/wiki/Wikimania>

9.3. Future Work

List of Figures

1.1	Research approach structure	4
2.1	RDF triple, image adapted from [CK04]	9
2.2	Same query from listing 2.2, visualized	11
2.3	Wikidata item about Douglas Adams (CC0 by Kritschmar) . . .	13
2.4	Wikidata item data representation in RDF (CC0 by Smalyshev)	15
3.1	Example of the list produced by the template in listing 3.1 . . .	18
3.2	Screenshot of a query built with SparqlFilterFlow [HLBE14] . .	21
3.3	Screenshot of a simple VizQuery query	22
3.4	Screenshot of an advanced VizQuery query	22
4.1	SPARQL feature usage in German and English Wikipedia Listeria queries (1252 total)	28
4.2	Graph patterns in SPARQL queries of the German and English Wikipedia Listeria queries (1252 total)	29
4.3	Most commonly occurring properties in property paths in SPARQL queries of the German and English Wikipedia Listeria queries; Showing only those with 10 or more occurrences	30
4.4	Usage of Wikidata-specific features in German and English Wikipedia Listeria queries (1252 total)	31
4.5	Initial search in Freebase Parallax (picture from [HK09]	35
4.6	Faceted search in ExConQuer (picture adapted from [AOA] . . .	36
4.7	Linked Data as a spreadsheet (picture adapted from [HGVS14] .	36
4.8	SPARQL query as a graph (picture adapted from [HLSE15] . .	37
4.9	SPARQL query as a tree (picture adapted from [AMH ⁺ 10] . . .	39
4.10	Screenshot of Sparklis [Fer14] from http://www.irisa.fr/LIS/ferre/sparklis/	39
4.11	Screenshot of the Linked Data Query Wizard’s interface element for adding predicates	41
4.12	Sparklis query with focus on the predicate of a triple	42
4.13	Screenshot of the query built using the Visual SPARQL Builder to assess the interaction effort	43
5.1	Statement filter with property ”instance of (subclass of)” and value ”City” to find all city-items in Wikidata	48
5.2	Two nested statement filters; This query would result in a list of items of female government leaders	50

List of Figures

5.3	Statement filter with a qualifier filter; This query would find any item that has a head of government with any value, and no qualifier for end date (i.e. that is the current head of government)	50
5.4	Mockup of a finished query generating a list of female mayors	52
5.5	Wikidata Query Builder "List of volcanoes in Japan"	54
5.6	Special item values as part of the item selector	55
5.7	Solution to the special property problem – a hint describing the default option	56
5.8	Original location of the "add qualifier filter" button	57
5.9	Qualifier section collapsed by default	57
5.10	Smaller qualifier button moved to the right side	58
6.1	Data flow between external services and the Query Builder	59
6.2	Query Builder component hierarchy during the creation of a complex query	61
6.3	Simplified Vuex data flow	62
6.4	Query Builder user interface state that produces the query in listing 6.3	65
7.1	UEQ categories, scales, and items (picture from [Sch15])	70
7.2	Distribution of participants' task completion status per task	73
7.3	UEQ results for the 6 scales, visualized by the provided data analysis tool	74
9.1	SPARQL feature usage in Listeria queries (total of 992 English, 669 German)	93
9.2	iSPARQL assessment	94
9.3	Linked Data Query Wizard assessment	95
9.4	QueryVOWL assessment	95
9.5	Sparklis assessment	95
9.6	SparqlFilterFlow assessment	96
9.7	VizQuery assessment	96
9.8	Visual SPARQL Builder assessment	96
9.9	Wikidata Query Builder assessment	97
9.10	Task 1 example item	97
9.11	Task 2 example item	98
9.12	Task 3 example items	99

List of Tables

3.1	Analysis of Wikidata Query Service example queries, adapted from Hernández et al. [HHR ⁺ 16]	19
3.2	LSQ SPARQL feature analysis, adapted from Saleem et al. [SAH ⁺ 15]	20
4.1	Feature-specific attributes	32
4.2	Interface-specific attributes	32
4.3	Interaction-specific attributes	33
4.4	Assessed query builder tools	34
4.5	Query Builder Tools Assessment	44
7.1	Questionnaire results per scale with comparison to results from the UEQ benchmark	75
7.2	Standard deviation and confidence intervals ($p = 0.05$) per scale	75
9.1	Assessed query builder tools with URLs	94

List of Tables

Bibliography

- [ABK⁺07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.
- [AMH⁺10] Oszkár Ambrus, Knud Möller, Siegfried Handschuh, et al. Konduit vqb: a visual query builder for sparql on the social semantic desktop. In *Workshop on visual interfaces to the social and semantic web*, 2010.
- [AMKP04] Chadia Abras, Diane Maloney-Krichmar, and Jenny Preece. User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, 37(4):445–456, 2004.
- [AOA] Judie Attard, Fabrizio Orlandi, and Sören Auer. Exconquer: Lowering barriers to rdf and linked data re-use. *Semantic Web*, (Preprint):1–15.
- [B⁺96] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [BE06] Jethro Borsje and Hanno Embregts. Graphical query composition and natural language processing in an rdf visualization interface. *Erasmus School of Economics and Business Economics, Vol. Bachelor. Erasmus University, Rotterdam*, 2006.
- [BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.
- [BL06] Tim Berners-Lee. Linked data-design issues. <http://www.w3.org/DesignIssues/LinkedData.html>, 2006.
- [CK04] Jeremy J Carroll and Graham Klyne. Resource description framework (rdf): Concepts and abstract syntax. 2004.
- [cro] Api:cross-site requests - mediawiki. https://www.mediawiki.org/wiki/API:Cross-site_requests. Accessed: 2017-12-20.
- [EGK⁺14] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. Introducing wikidata to the linked data web. In *International Semantic Web Conference*, pages 50–65. Springer, 2014.

Bibliography

- [Eip15] Lukas Eipert. Metadatenextraktion und vorschlagssysteme im visual sparql builder. 2015.
- [Fer14] Sébastien Ferré. Sparklis: a sparql endpoint explorer for expressive question answering. In *Proceedings of the 2014 International Conference on Posters & Demonstrations Track-Volume 1272*, pages 45–48. CEUR-WS. org, 2014.
- [Fin10] Kraig Finstad. The usability metric for user experience. *Interacting with Computers*, 22(5):323–327, 2010.
- [fre] Freebase shutdown blog post. <https://plus.google.com/u/0/109936836907132434202/posts/bu3z2wVqcQc>. Accessed: 2017-12-30.
- [Gar10] Jesse James Garrett. *Elements of user experience, the: user-centered design for the web and beyond*. Pearson Education, 2010.
- [hela] Help:items. <https://www.wikidata.org/wiki/Help:Items>. Accessed: 2017-12-30.
- [helb] Help:qualifiers. <https://www.wikidata.org/wiki/Help:Qualifiers>. Accessed: 2017-12-30.
- [helc] Help:ranking. <https://www.wikidata.org/wiki/Help:Ranking>. Accessed: 2017-12-30.
- [held] Help:sources. <https://www.wikidata.org/wiki/Help:Sources>. Accessed: 2017-12-30.
- [hele] Help:statements. <https://www.wikidata.org/wiki/Help:Statements>. Accessed: 2017-12-30.
- [HGVS14] Patrick Hoeffler, Michael Granitzer, Eduardo E Veas, and Christin Seifert. Linked data query wizard: A novel interface for accessing sparql endpoints. In *LDOW*, 2014.
- [HHR⁺16] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. Querying wikidata: comparing sparql, relational and graph databases. In *International Semantic Web Conference*, pages 88–103. Springer, 2016.
- [HK09] David F Huynh and David Karger. Parallax and companion: Set-based browsing for the data web. In *WWW Conference. ACM*, page 61, 2009.
- [HLBE14] Florian Haag, Steffen Lohmann, Steffen Bold, and Thomas Ertl. Visual sparql querying based on extended filter/flow graphs. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, pages 305–312. ACM, 2014.

- [HLSE15] Florian Haag, Steffen Lohmann, Stephan Siek, and Thomas Ertl. Queryvowl: Visual composition of sparql queries. In *International Semantic Web Conference*, pages 62–66. Springer, 2015.
- [Hol05] Andreas Holzinger. Usability engineering methods for software developers. *Communications of the ACM*, 48(1):71–74, 2005.
- [isp] Oat interactive sparql (isparql) query builder. <http://wikis.openlinksw.com/OATWikiWeb/InteractiveSparqlQueryBuilder>. Accessed: 2017-12-06.
- [kaf] Generating article placeholders from wikidata for wikipedia: Increasing access to free and open knowledge.
- [LFH17] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.
- [LHS08] Bettina Laugwitz, Theo Held, and Martin Schrepp. Construction and evaluation of a user experience questionnaire. In *Symposium of the Austrian HCI and Usability Engineering Group*, pages 63–76. Springer, 2008.
- [lisa] Wikipedia:vandalismmeldung/archiv/2015/09/12, howpublished = https://de.wikipedia.org/wiki/Wikipedia:vandalismmeldung/archiv/2015/09/12#benutzer:listeriabot_.28erl..29, note = Accessed: 2017-12-20.
- [lisb] Überlistet, howpublished = <http://magnusmanske.de/wordpress/?p=301>, note = Accessed: 2017-12-05.
- [ND86] Donald A Norman and Stephen W Draper. User centered system design. *Hillsdale, NJ*, pages 1–2, 1986.
- [nie] 10 usability heuristics for user interface design. <https://www.nngroup.com/articles/ten-usability-heuristics/>. Accessed: 2017-12-19.
- [Nor13] Don Norman. *The design of everyday things: Revised and expanded edition*. Basic Books (AZ), 2013.
- [PS⁺06] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.
- [PTVS⁺16] Thomas Pellissier Tanon, Denny Vrandečić, Sebastian Schaffert, Thomas Steiner, and Lydia Pintscher. From freebase to wikidata: The great migration. In *Proceedings of the 25th international conference on world wide web*, pages 1419–1428. International World Wide Web Conferences Steering Committee, 2016.

Bibliography

- [rdf] Wikibase/indexing/rdf dump format, howpublished = https://www.mediawiki.org/wiki/wikibase/indexing/rdf_dump_format, note = Accessed: 2017-12-08.
- [RSBS08] Alistair Russell, Paul R Smart, Dave Braines, and Nigel R Shadbolt. Nitelight: A graphical tool for semantic query construction. 2008.
- [SAH⁺15] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Lsq: the linked sparql queries dataset. In *International Semantic Web Conference*, pages 261–269. Springer, 2015.
- [Sch15] Martin Schrepp. User experience questionnaire handbook. *All you need to know to apply the UEQ successfully in your project*, 2015.
- [SKW07] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.
- [Sny03] Carolyn Snyder. *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann, 2003.
- [spa] [wikidata] announcing the release of the wikidata query service. <https://lists.wikimedia.org/pipermail/wikidata/2015-September/007042.html>. Accessed: 2017-12-05.
- [STP14] Dominik Schweiger, Zlatko Trajanoski, and Stephan Pabinger. Sparqlgraph: a web-based platform for graphically querying biological semantic web databases. *BMC bioinformatics*, 15(1):279, 2014.
- [sum] Summative usability testing | usability body of knowledge. <http://www.usabilitybok.org/summative-usability-testing>. Accessed: 2017-12-22.
- [SZ] Timo Stegemann and Jürgen Ziegler. Pattern-based analysis of sparql queries from the lsq dataset.
- [tog] First principles of interaction design (revised & expanded). <http://asktog.com/atc/principles-of-interaction-design/>. Accessed: 2017-12-19.
- [Tun09] Daniel Tunkelang. Faceted search. *Synthesis lectures on information concepts, retrieval, and services*, 1(1):1–80, 2009.
- [viz] Vizquery, howpublished = <https://tools.wmflabs.org/hay/vizquery/>, note = Accessed: 2017-12-12.

- [VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.
- [vue] What is vuex? · vuex. <https://vuex.vuejs.org/en/intro.html>. Accessed: 2017-12-21.
- [wdq] Wdqs usage dashboard. http://discovery.wmflabs.org/wdqs/#wdqs_usage. Accessed: 2017-12-30.
- [web] Web components | mdn. https://developer.mozilla.org/de/docs/Web/Web_Components. Accessed: 2017-12-20.
- [wika] List of wikipedias. https://meta.wikimedia.org/wiki/List_of_Wikipedias. Accessed: 2017-12-13.
- [wikb] Template:wikidata list, howpublished = https://en.wikipedia.org/wiki/template:wikidata_list, note = Accessed: 2017-12-05.
- [wikc] Wikidata:introduction. <https://www.wikidata.org/wiki/Wikidata:Introduction>. Accessed: 2017-12-30.
- [wikd] Wikimedia ui style guide. <https://wikimedia.github.io/WikimediaUI-Style-Guide/>. Accessed: 2017-12-19.
- [wike] The wikipedia data revolution, howpublished = <https://blog.wikimedia.org/2012/03/30/the-wikipedia-data-revolution/>, note = Accessed: 2017-12-05.
- [wikf] Wikipedia:bots. <https://en.wikipedia.org/wiki/Wikipedia:Bots>. Accessed: 2017-12-30.
- [wikg] wikipedia.org traffic statistics. <https://www.alexa.com/siteinfo/wikipedia.org>. Accessed: 2017-12-30.

Appendix

SPARQL Analysis

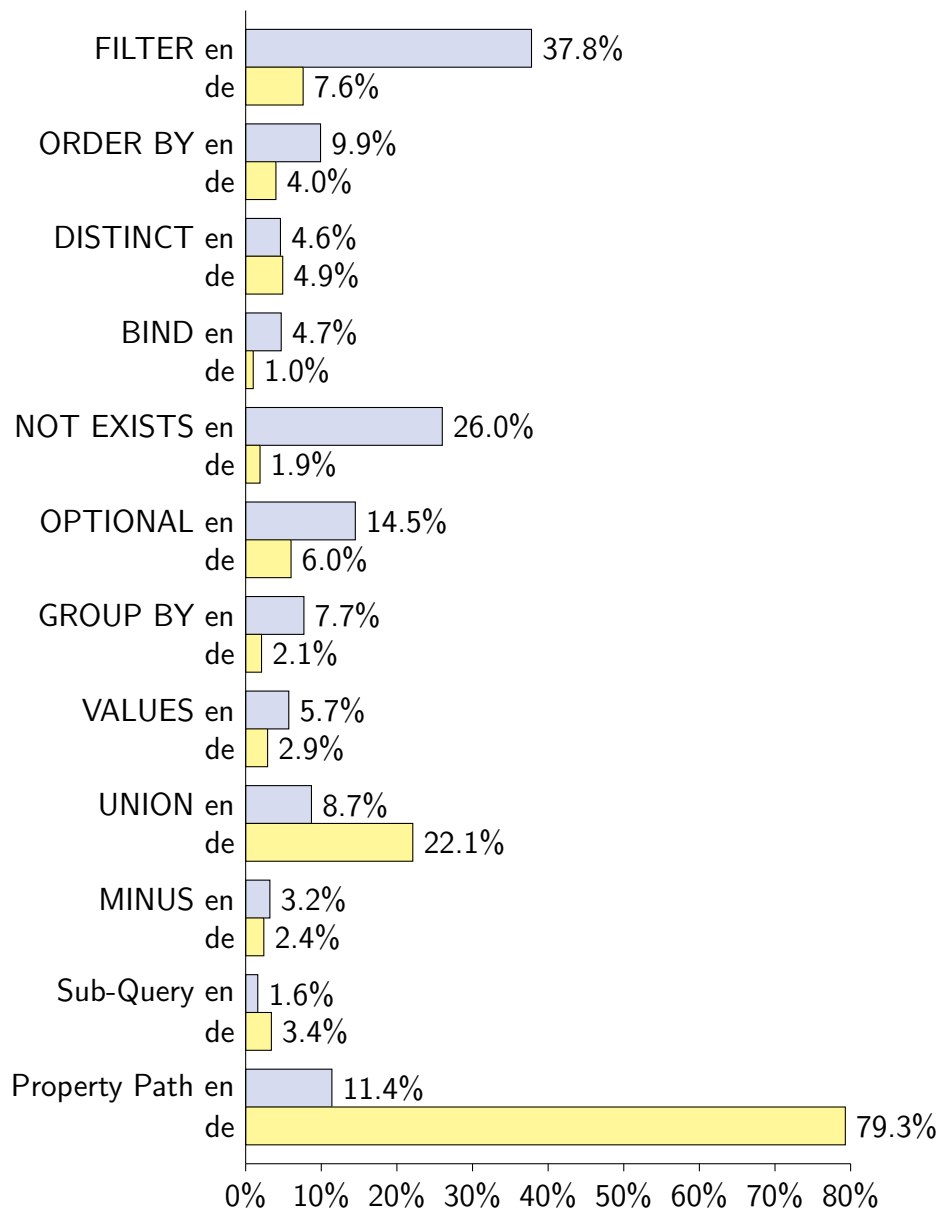


Figure 9.1: SPARQL feature usage in Listeria queries (total of 992 English, 669 German)

Tools Analysis

Project Name	Category	Web Demo	Source
ExConQuer	EF	-	[AOA]
Freebase Parallax	EF	-	[HK09]
iSPARQL	TG	http://dbpedia.org/isparql/	[isp]
Konduit VQB	TT	-	[AMH ⁺ 10]
LDQW ^a	ES	https://code.know-center.tugraz.at/search	[HGVS14]
NITELIGHT	TG	-	[RSBS08]
QueryVOWL	TG	http://vowl.visualdataweb.org/queryvowl/queryvowl.html	[HLSE15]
Sparklis	TT	http://www.irisa.fr/LIS/ferre/sparklis/	[Fer14]
SparqlFilterFlow	TF	http://sparql.visualdataweb.org/sparqlfilterflow.php	[HLBE14]
SPARQLGraph	TG	http://sparqlgraph.i-med.ac.at/	[STP14]
SPARQLViz	TU	-	[BE06]
Visual SPARQL Builder	TG	http://leipert.github.io/vsb/dbpedia/	[Eip15]
VizQuery	TU	https://tools.wmflabs.org/hay/vizquery/	[viz]

Table 9.1: Assessed query builder tools with URLs

^aLinked Data Query Wizard

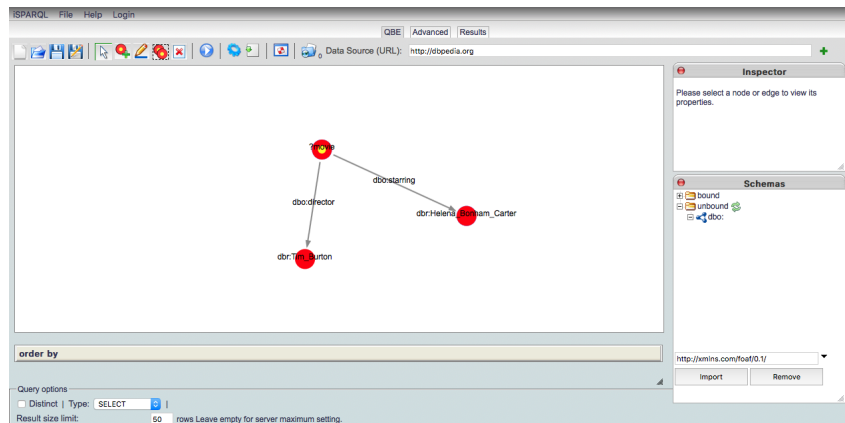


Figure 9.2: iSPARQL assessment

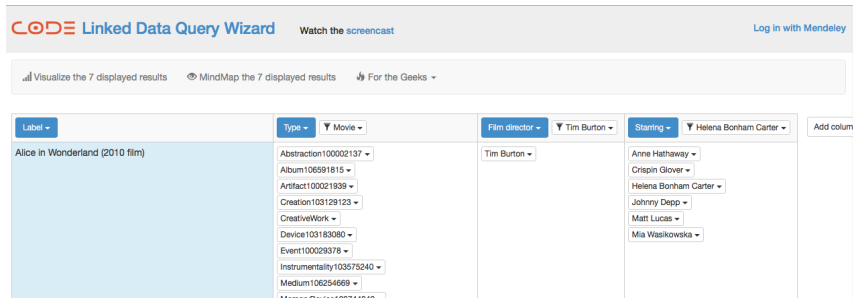


Figure 9.3: Linked Data Query Wizard assessment

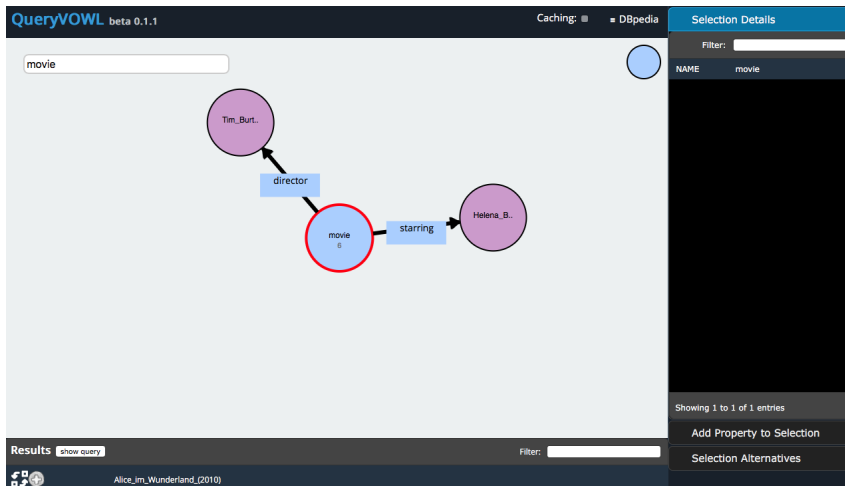


Figure 9.4: QueryVOWL assessment

Figure 9.5: Sparklis assessment

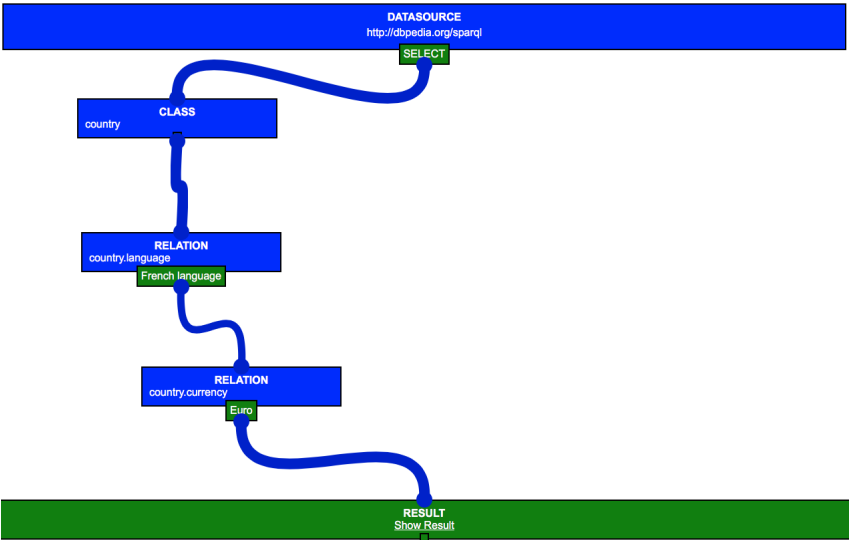


Figure 9.6: SparqlFilterFlow assessment

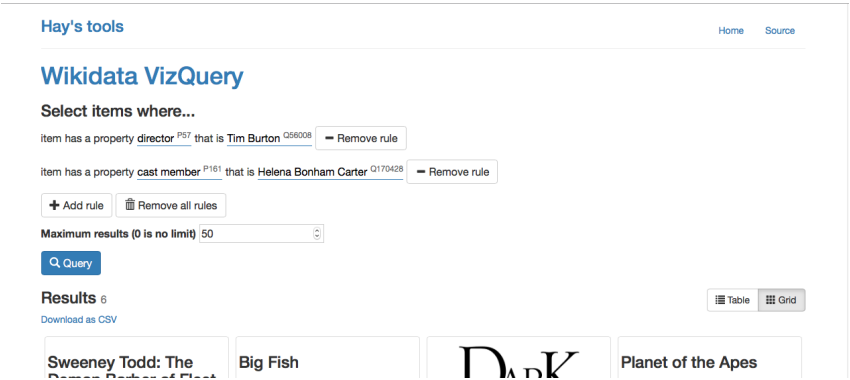


Figure 9.7: VizQuery assessment

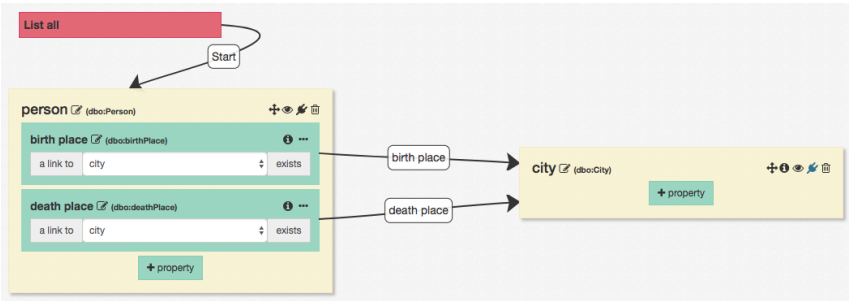


Figure 9.8: Visual SPARQL Builder assessment

Wikidata Query Builder

Create lists of items by filtering them based on their statements Examples ▾

Property

director ▾

Value

Specific item ▾

Tim Burton ▾

×

+ add qualifier filter

Property

cast member ▾

Value

Specific item ▾

Helena Bonham Carter ▾

×

+ add qualifier filter

+ add statement filter

▸ Result columns

▸ List options

Show Results

Show Query

Share Query

item	itemLabel
wd:Q323318	Dark Shadows
wd:Q212041	Sweeney Todd: The Demon Barber of Fleet Street

Figure 9.9: Wikidata Query Builder assessment

User Study Tasks

Socks (Q1371145)

cat belonging to Bill Clinton and family edit

Socks Clinton

[In more languages](#) [Configure](#)

Language	Label	Description	Also known as
English	Socks	cat belonging to Bill Clinton and family	Socks Clinton
German	Socks	Katze von Bill Clinton und seiner Familie	
French	Socks	Chat appartenant à Bill Clinton et sa famille	
Bavarian	No label defined	No description defined	

[All entered languages](#)

Statements

instance of

cat

edit

0 references

+ add reference

Figure 9.10: Task 1 example item

Bláhnjúkur

(Q886940)

mountain in Iceland

Blahnjukur

edit

▼ In more languages

Configure

Language	Label	Description	Also known as
English	Bláhnjúkur	mountain in Iceland	Blahnjukur
German	Bláhnúkur	No description defined	Bláhnjúkur
French	Bláhnjúkur	montagne islandaise	Blahnjukur
Bavarian	No label defined	No description defined	

All entered languages

Statements

instance of

mountain

edit

► 1 reference

volcano

edit

▼ 0 references

+ add reference

+ add value

country

Iceland

edit

▼ 0 references

+ add reference

+ add value

Figure 9.11: Task 2 example item

Barcelona

(Q1492)

capital of Catalonia and second largest city in Spain

Barcelona, Spain

edit

▼ In more languages

Configure

Language	Label	Description	Also known as
English	Barcelona	capital of Catalonia and second largest city in Spain	Barcelona, Spain
German	Barcelona	Hauptstadt Kataloniens und zweitgrößte Stadt Spaniens	
French	Barcelone	capitale de la communauté autonome de Catalogne (Espagne)	
Bavarian	Barcelona	No description defined	

All entered languages

Statements

instance of

city

edit

▼ 0 references

+ add reference

head of government

Ada Colau

start time

13 June 2015

edit

▼ 0 references

+ add reference

Ada Colau

(Q4779594)

Spanish activist

edit

▼ In more languages

Configure

Language	Label	Description	Also known as
English	Ada Colau	Spanish activist	
German	Ada Colau	katalanische Aktivistin und Politikerin	
French	Ada Colau	femme politique espagnole	Ada Colau Ballano
Bavarian	No label defined	No description defined	

All entered languages

sex or gender

female

edit

► 1 reference

+ add value

Figure 9.12: Task 3 example items