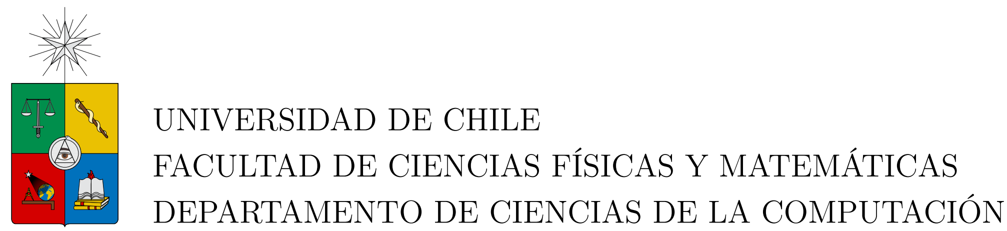

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A FACETED BROWSING INTERFACE FOR DIVERSE LARGE-SCALE RDF DATASETS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN
CIENCIAS MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JOSÉ IGNACIO MORENO VEGA

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
BENJAMÍN BUSTOS CÁRDENAS
CARLOS BUIL ARANDA

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE: Ingeniero Civil en Computación
y grado de Magister en Ciencias Mención Computación
POR: José Ignacio Moreno Vega
FECHA: 27 de junio de 2018
PROFESOR GUÍA: Aidan Hogan

A FACETED BROWSING INTERFACE FOR DIVERSE LARGE-SCALE RDF DATASETS

Las bases de conocimiento en RDF contienen información acerca de millones de recursos, las cuales son consultadas utilizando el lenguaje estándar de consultas para RDF: SPARQL. Sin embargo, esta información no está accesible fácilmente porque requiere conocer el lenguaje SPARQL y la estructura de los datos a consultar; requisitos que no cumple un usuario común de internet.

Se propone una interfaz de navegación por facetas para estos datos de gran tamaño que no requiere conocimientos previos de la estructura ni de SPARQL. La navegación por facetas consiste en agregar filtros (conocidos como facetas) para mostrar únicamente los elementos que cumplen los requisitos. Interfaces de navegación por facetas para RDF existentes no escalan bien para las bases de conocimientos actuales.

Se propone un nuevo sistema que crea índices para búsquedas fáciles y rápidas sobre los datos, permitiendo calcular y sugerir facetas al usuario. Para validar la escalabilidad y eficiencia del sistema, se escogió Wikidata como la base de datos de gran tamaño para realizar los experimentos de desempeño. Luego, se realizó un estudio de usuarios para evaluar la usabilidad e interacción del sistema, los resultados obtenidos muestran en qué aspectos el sistema desempeña bien y cuáles pueden ser mejorados. Un prototipo final junto a un cuestionario fue enviado a contribuidores de Wikidata para descubrir como este sistema puede ayudar a la comunidad.

Abstract

RDF knowledge bases contain information about millions of resources, which can be queried using the standard querying language for RDF: SPARQL. However, this information is not easily accessible because it requires knowledge of SPARQL and the structure of the specific dataset to query; requirements that a common Internet user would not met.

We propose a faceted browsing interface for these large-scale datasets that does not require prior knowledge of their structure or SPARQL. Faceted browsing consists of adding filters (known as facets) to display only the elements that meet the restrictions. Existing faceted browsing interfaces for RDF do not scale well for current knowledge bases.

We propose a new system that creates indexes for fast and easy searches over the dataset, allowing to compute and suggest facets to the user. To validate the scale and efficiency of the system, Wikidata was chosen as a large-scale dataset over which to conduct performance experiments. Then, a user study was performed to evaluate the usability and responsiveness of the system, obtaining results that shows in which aspects the system performs well and which ones can be improved. A final prototype and questionnaire was submitted to the Wikidata contributors to see how this system could help the community.

A mi familia y amigos

Contents

Abstract **ii**

1 Introduction **1**

1.1 Objectives . . . 2

1.1.1 Main objective . . . 2

1.1.2 Specific objectives . . . 2

1.2 Methodology . . . 2

1.3 Expected Results . . . 3

1.4 Outline of this Document . . . 4

2 Background **5**

2.1 Semantic Web . . . 5

2.1.1 Data Model . . . 6

2.1.2 Queries on RDF . . . 9

2.1.3 Linked Data . . . 9

2.1.4 Wikidata . . . 10

2.1.5 Query Interfaces . . . 12

2.2 Information Retrieval . . . 15

2.2.1 Inverted Index . . . 15

2.2.2 Ranking Importance . . . 15

3 System Overview **17**

3.1 User Interactions . . . 17

3.1.1 Keyword . . . 19

3.1.2 Type . . . 19

3.1.3 Results . . . 19

3.1.4 Facets . . . 20

3.2 Query Expressiveness . . . 21

3.2.1 Operations . . . 21

3.2.2 Analysis versus SPARQL Queries . . . 22

4 Back end 24
4.1 Initial index 24
4.1.1 Structure of the documents 24
4.1.2 Building the index 26
4.2 Ranking data 27
4.3 Types index 28
4.4 Queries and results 28
4.4.1 Queries by identifier 29
4.4.2 Queries by keyword 29
4.4.3 Queries by type 29
4.4.4 Queries by property-value 30
4.4.5 Mixed queries 30
4.5 Improving times and caching 31
4.6 Cache indexes 34
4.6.1 Cache for properties 34
4.6.2 Cache for values 34

5 Front end 36
5.1 Home page 36
5.2 Results page 38
5.2.1 Navigation bar 39
5.2.2 Results entries 39
5.2.3 Facet selection 40

6 Evaluation 41
6.1 System Performance 41
6.1.1 Indexes statistics 41
6.1.2 Browsing performance 43
6.2 User study 44
6.3 Wikidata's collaborators study 48
6.4 Discussion 50

7 Conclusion 51
7.1 Current limitations 51
7.2 Future work 52

Bibliography 54

List of Tables

4.1 Times of processing results for most common types 31

6.1 Times of all index-creation steps 42
6.2 Size and documents stored by index 42
6.3 Queries with one result based on the number of facets 44
6.4 Tasks results of the user study 46

List of Figures

2.1 Semantic Web Stack 6
2.2 RDF graph 8
2.3 Data model in Wikidata (Image from Wikidata) 11
2.4 Facete interface 13
2.5 SemFacet interface 14

3.1 Flow of user interactions 18

4.1 Ratio of values that need cache versus threshold 33

5.1 Home page 36
5.2 Type selection 37
5.3 Search by keyword Tab 38
5.4 Results page 38
5.5 Value selection for a property 40

6.1 Times to load the Results page 43
6.2 Times to load values for a property 45
6.3 Wikidata Query Service 46
6.4 Responsiveness score according to users 47
6.5 Usability score according to users 48
6.6 Answers of Wikidata's collaborators questionnaire 49

Chapter 1

Introduction

In the world of today when someone wants to search for anything, the first place to look up is the Web. The Web contains a lot of information and every day this information is increasing. Most of this information is in plain text, but there is a problem: machines usually do not understand natural language, so to search for something, it has to be by matching keywords. Overcoming this problem is the idea behind the Semantic Web.

The Semantic Web's goal is to structure the existing data on the Web in a machine readable format in order to allow interoperability between websites. With this goal, the Semantic Web uses the RDF standard, which stores data in triples of the form subject, predicate and object. Defining best practices about how datasets expressed in RDF should relate to one another on the Web is known as Linked Data.

There are various large datasets of RDF on the Web, like DBpedia and Wikidata. These datasets contain hundred of millions of triples; given that these datasets are structured, they can be used to answer complex queries over information drawn from many sources. However these datasets cover many different topics and contain data contributed by thousands of different editors. Because of this, such datasets do not have a clear schema or defined structure. This causes a first level of complication for users wishing to query these data.

The current standard to make queries over RDF data is the query language SPARQL. Here is where the problem mentioned earlier takes place: over large-scale data it is difficult to find a common structure in the data, which complicates querying. Furthermore, in order to obtain good results, it is necessary to learn how to write queries in the SPARQL language, as well as having knowledge of the structure of the data one wants to retrieve. Most users on the Web do not satisfy these requirements and thus cannot query these datasets, reducing their value since often, it is too difficult for such users to answer interesting questions over such data.

This work presents an interface to search over such datasets (specifically over Wikidata, though the methods presented generalize to other datasets) using *faceted browsing*. Faceted browsing consists in making a general search over the data and then adding filters known as

facets (characteristics of the element to find) to refine the results. This kind of interface is very common on popular websites such as online shops (like Amazon): while searching for a product you can add facets like brand, price or year to find the one you want.

While faceted queries do not cover SPARQL, they offer an intuitive and flexible query mechanism. There are currently interfaces that allow faceted browsing over RDF but these do not support a large dataset like Wikidata with hundreds of millions of triples, where the number of possible facets is immense, or the data can be in multiple languages. A main goal of designing such a faceted browsing interface is to unlock the potential value of large-scale RDF datasets available on the Web (like Wikidata and DBpedia) by helping common users pose queries against them.

1.1 Objectives

1.1.1 Main objective

The main objective of this thesis work is to create a new interface for faceted browsing that supports a large dataset (hundreds of millions of triples). The data could be about different topics with multiple properties, with incomplete or missing information and be in multiple languages. That way, it would be possible to search and pose (simple) queries over diverse, large-scale, multilingual RDF datasets with no previous knowledge of its structure nor SPARQL.

1.1.2 Specific objectives

- Structure and index the data making it possible to query the dataset with facets and also keywords.
- Allow the possibility for a universal system that supports any kind of RDF dataset.
- Support content in different languages for keyword searches.
- Compute and display the important facets given a group of data or partial results.
- Develop ranking mechanisms to display the most important facets and entities first.
- Make the system available as a Web application with a responsive and intuitive interface.

1.2 Methodology

To accomplish the previous objectives, the work will be done following these steps in order.

- Experiment and test with existing faceted browsing systems over large datasets like Wikidata and see how well or badly these systems support such a dataset.
- Select an appropriate indexing scheme to support the necessary user interactions (e.g. keyword search and auto complete function).
- Create the index with a structure that enables search using facets.
- Develop the core of the system with a basic search interface to compute the facets and filter the results with a given facet.
- Improve the usability of the system allowing ranking of the results and facets according to measures of relevance and centrality.
- Do performance tests over the system based on computation time versus complexity of the query.
- Perform a user study to measure usability: users perform queries with this system versus other systems and answer some questions.

1.3 Expected Results

The final product of this work is an application for faceted browsing over multi-domain, diverse, large-scale and incomplete RDF datasets.

This application should be available as an online service, allowing to do faceted browsing over Wikidata. The service should (i) offer responsive runtimes, (ii) offer suitable rankings of facets and results, (iii) offer keyword search and autocomplete textual features, (iv) support various languages, (v) avoid facets generating empty results.

Building such an application for large-scale datasets like Wikidata is challenging and requires exploring novel techniques to handle billions of triples while maintaining responsive runtimes. Another expected result of this work is a set of algorithms by which such a system can be realized. A particularly challenging aspect is to compute exact facets over millions of results, capturing all (and only) restrictions that will give non-empty results. An important result of this work will be a novel caching method to achieve these goals.

We will also provide experimental results that validate that the proposed system meets the above criteria, in particular, the responsiveness of the system (allowing interactive browsing and search), and the general usability of the system (with a user study).

1.4 Outline of this Document

Chapter 2 will talk about the Semantic Web, what is RDF and explain the background of this work to understand the next chapters. Chapter 3 gives an overview of the system. Chapters 4 and 5 explain the development of the system starting from the data itself until the final system is complete. Chapter 6 presents the evaluation results obtained with the system. Chapter 7 concludes this work explaining limitations of the system and discussing future work.

Chapter 2

Background

This work covers two major topics in Computer Science: Semantic Web and Information Retrieval. The Semantic Web aims to add semantics to the content of the Web, while Information Retrieval is the area in charge of obtaining relevant information from a collection of information resources (typically, textual documents).

2.1 Semantic Web

Nowadays, the Web is growing rapidly due to a large user base generating more and more content every day. Due to this large amount of data, it is necessary for machines to process this content to allow users to find relevant information when needed. Search engines like Google thus crawl and index large portions of the Web; however, machines usually do not "understand" the content of the Web itself. Rather the machine processes text strings, that may be separated into words by a delimiter, and compute statistics about for example, the frequencies of particular strings in a document. In order to make a machine understand a message, it needs a defined structure and semantics.

The Semantic Web's main objective is to allow the content of the Web to be represented in a machine readable format, defining the structure, data model and semantics for each piece of information [2]. With this idea in mind, the Semantic Web has defined an architecture known as the *Semantic Web Stack*, formed of layers, where each layer depends on the ones below it (see Figure 2.1).

The bottom layer has identifiers and characters; these allow to assign to each resource being described a unique ID using the available characters. The character set accepted in this layer is the *Unicode* standard because it supports multiple languages and special symbols. The identifier scheme used is *IRI* (Internationalized Resource Identifier), a variant of *URI* (Uniform Resource Identifier) used widely on the Web, allowing more characters from Unicode. Right above this layer, there is the *Syntax* layer: it defines how the identifiers structure a grammar; here there are the XML and JSON formats, widely used already, but

it also includes *Turtle* and *N-Triples* formats made specially for *RDF triples*; the core of the next layer: Data Model, which will be explained in the next section.

The next level has two components: Schema & Ontologies and Querying. Schema & Ontologies are explained in Section 2.1.1, and Querying in Section 2.1.2. The following layers, seen in the figure in red boxes, are theoretical layers with no standard defined, but with a defined proof of concept. Unifying Logic aims to combine ontological reasoning and querying. The Proof layer aims to provide a *proof* of the procedure or information used to the client, and the Trust layer should determine which services have access to which data. The Cryptography layer is to the side of all others because all parts of the Semantic Web would require cryptography techniques for verifying identity and access control for sensitive data.

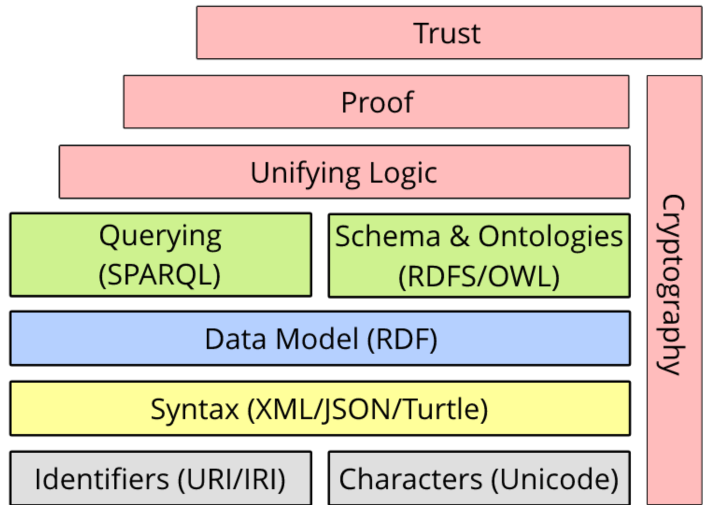

Figure 2.1: Semantic Web Stack

2.1.1 Data Model

Resource Description Framework (RDF)

The RDF standard [11] is the most important component of the Semantic Web; its mission is to provide the core data model for structuring the factual content of the Web. With a data model well defined, it is possible to accomplish interoperability between data published by different websites. The standard is composed of RDF *terms*, *triples*, *graphs* and *vocabulary*.

RDF terms are the minimal piece of information on the Semantic Web: a term represents a resource that can be an IRI, a literal value or a blank node. IRIs, as explained before, are an identifier for a resource on the internet. Literal values can be a text string with or without an associated language (for example, the "`@en`" tag represents that the text is in English)

or a datatype such as an integer number, a boolean, etc. Blank nodes are used to represent that something exists but its identifier or value is unknown.

With RDF terms, it is possible to make a sentence that relates terms. This is done grouping three terms in a specific order forming a *triple*: as the name suggests an RDF triple is a 3-tuple of RDF terms. The first term is the *subject*, which indicates what resource we are talking about, where this term can be an IRI or a blank node; the second one is the *predicate* which has to be an IRI, establishing a relationship between the subject and the third term: the *object*, which can be an IRI, blank node or a literal value. A triple is the core of the Semantic Web data model because a triple is a fact, for example, *Santiago* is the *capital city* of *Chile*.

Example 2.1. This is a small RDF dataset based on Wikidata triples. Note that IRIs are not accurate for the sake of understanding since Wikidata uses numbers as identifiers.

```
ex:Earth ex:highestPoint ex:MountEverest
ex:MountEverest ex:continent ex:Asia
ex:Asia ex:partOf ex:Earth
ex:Asia ex:sharesBorderWith ex:Europe
ex:Europe ex:partOf ex:Earth
ex:MountEverest ex:namedAfter ex:GeorgeEverest
ex:GeorgeEverest ex:countryOfCitizenship ex:UnitedKingdom
ex:UnitedKingdom ex:continent ex:Europe
ex:GeorgeEverest ex:placeOfDeath ex:London
ex:UnitedKingdom ex:capital ex:London
```

Listing 2.1: Small RDF dataset

Note that `ex:` is a prefix shortcut indicating an IRI and its respective domain, such that `ex:Earth` refers to the IRI `http://example.org/Earth`.

Grouping multiple triples we have a graph where the subject and the object are the vertices and where the predicate labels the directed edge that connects them; this way it is possible to see the connections between all the different resources described on the Web. Over a graph we can compute interesting statistics and algorithms to process and understand how the data relates to one another.

Example 2.2. Figure 2.2 shows the RDF graph for the dataset in Listing 2.1.

Lastly, to add some sense of meaning or vocabulary to all of this, the RDF standard defines a group of IRIs with special meaning. The most important are `rdf:type` and `rdf:Property`. The first is to indicate the type of the subject, for example, *Chile* is a *country*, therefore its type is country; this allows to group the resources into *classes* such as the class of all countries or all people. The second term (`rdf:Property`) is the class of all the elements that go in

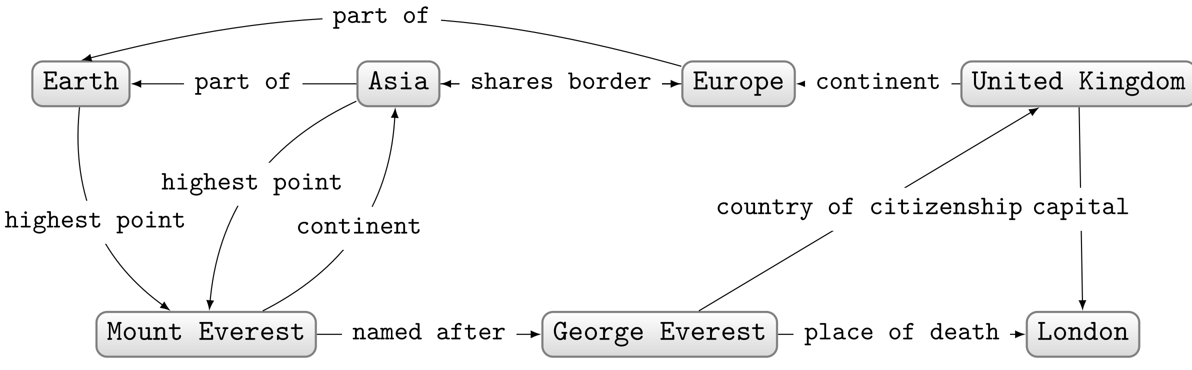

Figure 2.2: RDF graph

the predicate position of a triple, so they shall be known as *properties*; in other words, a triple contains a subject, property and value of that property. The other terms in the RDF vocabulary will be omitted as they are not relevant for this document.

RDF Schema and Web Ontology Language (OWL)

The RDF standard is a powerful tool to start structuring data on the web, where the RDF vocabulary gives a meaning to each element and an organization to these data. However, these definitions are not enough to define richer semantics and made necessary new standards that expand on the vocabulary already defined.

RDF Schema [3] was the first such standard and adds relations between classes and properties with, for example, `rdfs:subClassOf` and `rdfs:subPropertyOf`. This allows for inferring information that was previously unknown; for example: if we know that *Santiago de Chile* is a *City* and *City* is a sub class of *Settlement*, the machine can deduce that *Santiago de Chile* is a *Settlement* too, even if that triple is not in its data.

The Web Ontology Language (OWL) [9] expands this idea even more adding new vocabulary including `owl:equivalentClass`, `owl:equivalentProperty` and `owl:sameAs`, thus making it possible to integrate data from different resources, like that two resources have their own IRI for people, one is named *Person* and the other is *Human* so both are *equivalent classes*. OWL adds other expressions such as `owl:inverseOf` (like parent and child), `owl:SymmetricProperty` (like sibling) and more, all this to provide integration and completion of data, through reasoning.

Note that in the present work, we do not consider reasoning but focus on faceted browsing over explicit data. Hence we do not discuss RDFS and OWL in further detail.

2.1.2 Queries on RDF

The current standard to query over RDF data is SPARQL [5], which is similar to SQL (Structured Query Language) but applied to RDF triples. SPARQL uses only explicit triples: it cannot typically infer or deduce new information as discussed in the previous section. SPARQL syntax is based on the *Turtle* syntax of RDF (a common format to write RDF triples in subject-predicate-object order) and SPARQL keywords.

The most relevant part of a SPARQL query is the *query clause*, which is where the patterns of the triples are specified. The triples may contain variables and ask the query to return results where variables match terms in the data, preserving the structure of the query and the data.

Example 2.3. Using the dataset in Listing 2.1, a query to answer what is the highest point in the planet and in what continent is located, is constructed as follows: variables start with a question mark; the query will return the variables specified in the SELECT clause with all IRIs that match in the WHERE clause. This query will return the pair `ex:MountEverest, ex:Asia` because `?s1` matches in the WHERE clause with `ex:MountEverest` in the dataset and the same with `?s2` matching `ex:Asia`.

```
SELECT ?s1 ?s2
WHERE {
  ex:Earth ex:highestPoint ?s1 .
  ?s1 ex:continent ?s2
}
```

Listing 2.2: SPARQL query

In conclusion, in order to succesfully construct SPARQL queries it is necessary to know the RDF graph we are querying and the main structure of the data, like what IRIs are used for which classes or properties. This, however is a major problem when a user, who knows nothing about the dataset or even RDF at all, wants to query the data. In the last section of this chapter are examples of multiple interfaces to query over RDF that do not assume user expertise in SPARQL nor the structure of the dataset.

2.1.3 Linked Data

We have covered the main components of the Semantic Web, but where on the Web is all this *RDF data*? The most obvious connection between the two are the IRIs that look and sound similar to the URLs we type in web browsers. *Linked Data* is then the (best) way to deploy all these Web Semantic standards on the Web we all know. There are four principles for Linked Data [6].

1. Use IRIs to identify things.

2. Use HTTP IRIs to reference these things.

3. Return useful data when looking up about these IRIs, ideally in RDF. With the first three principles we can now get more structured information about something when looking up its IRI.

4. Include links to reference other documents by using its IRI, thus making the data *linked*.

With these guidelines, the community started developing sets of new Linked Data guidelines and publishing their data in this format. The most popular RDF datasets on the Web currently are DBpedia [8], Wikidata [14] and YAGO [13]; these describe resources (also known as entities) based on Wikipedia articles, thus giving a lot of data about millions of different subjects. With that many entries with diverse types, properties and domains, a major problem is how to enable non-experts users to effectively make queries that return the expected result and not require knowledge of SPARQL.

In the following, we describe the Wikidata dataset, published on the Web as Linked Data, which will be used later in this thesis.

2.1.4 Wikidata

Wikidata is a free, multilingual and open database collecting structured data that can be read and edited by humans and machines. Wikidata currently has over 40 million resources and over 350 million statements, with around 14 millions edits per month by over 30 thousand active editors.

Wikidata follows a specific rule to assign IRIs to resources. Wikidata's IRIs consist of a letter and a number; all properties start with the letter `P` and the other resources with the letter `Q`. Although it is not possible to know what the resource is by looking at its IRI, resources with the same name will not be confused; for example, the IRI for the planet Earth is `http://www.wikidata.org/entity/Q2`, abbreviated as `wd:Q2`, and for the element earth is `wd:Q2488752`.

To name the resource, a triple with the property `rdfs:label` is required where the value needs a language tag. From the previous example, the triple to give a name to `wd:Q2` is `wd:Q2 rdfs:label "Earth"@en`; it is possible to add more triples to add the label for different languages by changing the language tag.

Wikidata also uses two other properties to better describe the resource other than the label. These two properties are `skos:altLabel` to indicate other names the resource is known for (`wd:Q2 skos:altLabel "Planet Earth"@en`) and `schema:description` to describe it in words (`wd:Q2 schema:description "third planet from the Sun in the Solar`

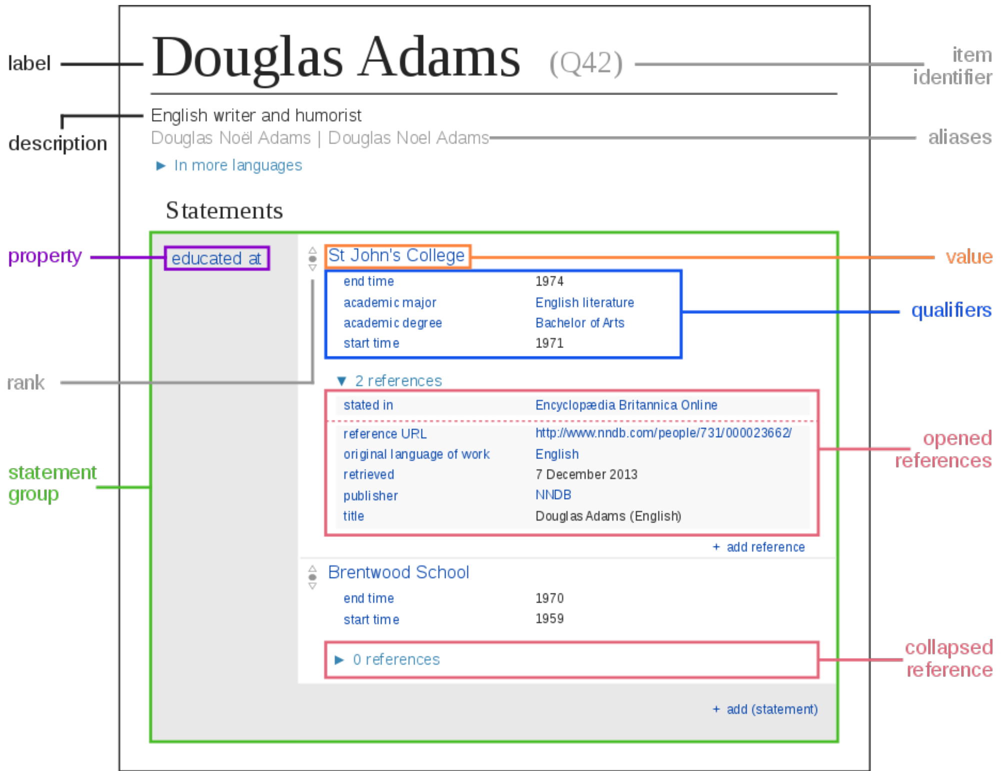

Figure 2.3: Data model in Wikidata (Image from Wikidata)

`System"@en`). These three properties will be used (if available) in the faceted browsing interface to help identify the resource to the users.

Wikidata's statements describe detailed characteristics of a resource and consist of a property and one or more values; it can also be expanded with qualifiers, references and ranks. Properties and values in Wikidata work the same way as in an RDF triple, where a value is another Wikidata resource, literal value or blank node. Qualifiers refine or further describe the value given to a property in a statement, for example, *Donald Trump* is the *U.S. president* but that statement has a *start time* qualifier of 20 January 2017. References allow to back up the data provided in the statement and ranks in statements establish the preferred value when a property has multiple values; for example, *Barack Obama* was the *U.S. president* but also a *senator*, however, the president value is preferred. This work does not use qualifiers, references or ranking of the statements, as their function is to expand on the statement but they are not an important part of it for initially querying entities; however, it would be interesting to extend this work in the future to consider such additional information.

2.1.5 Query Interfaces

There are multiple proposals to query RDF data (other than to write a SPARQL query) that aim to help users with limited knowledge of RDF and the Semantic Web. These include visualization of RDF graphs, Question Answering and Faceted Browsing. In these methods, the user does not require experience of anything about the Semantic Web (though it may be beneficial). Visualization over RDF is easy to understand because all RDF datasets are a graph. However, more specific visualizations are tied to certain types of entity like geographical data using a map and it is not easy to offer visualizations over larger datasets.

Question Answering consists in giving the correct answer to a question written in natural language; this is accomplished by translating the question to a structured query (e.g. SPARQL) to give an answer. The program needs to know what dataset is being queried and the existing properties and overall structure of that dataset, so approaches often require a training dataset where both questions and answers are given. This method is heavily dependant on the dataset used and the question types that it supports (certain languages, complexity of the question); also, systems may only answer questions similar to its training set. Examples of question answering interfaces are: QAKis[1] and AskWikidata[2].

Example 2.4. A question answering interface trained with the dataset in Listing 2.1 should be able to answer the question *"In which continent is the highest point on Earth located?"* The user directly types the question and the interface returns the correct answer: Asia.

Faceted Browsing consists in adding facets to an initial query (usually a type query, like all countries or all persons). Each facet is a property to filter the previous results. For example, a user may first select all countries and then filter the results with the facet of member of the United Nations. Then, by adding multiple facets, the user can find all results that match the specified criteria. This structure of searching content is mainly used in online shops (like Amazon) to filter its products by price, brand and other properties. This method requires minimal knowledge of the structure of the dataset being used and it can only return results based on direct properties: it is not possible to specify properties of a given facet; for example: it is not possible to find all presidents of countries that are members of the UN, since the search is for presidents but the facet *member of the UN* applies to the country, not to the president. For this reason, faceted browsing interfaces cannot be used to express complex queries like SPARQL. However, they are familiar to many users through interfaces like Amazon. Given that they are considered user friendly, a number of such interfaces have already been defined for RDF datasets; such interfaces include gFacet [7], Facete [12] and SemFacet [1]. However, these systems do not support large and diverse dataset about different

[1]http://qakis.org/
[2]http://tools.wmflabs.org/bene/ask/

topics with hundreds of possible facets such as the Wikidata dataset. We now discuss these systems in more detail.

gFacet is a combination of faceted browsing and graph-based visualization, where the user selects a category of the data based on a keyword search and then, the system displays a node with the list of all items in the category and their respective facets. Additional nodes on the visualization are added by selecting a facet from another category, thus making an edge between such nodes. When one or more items in a category are selected, the other nodes will highlight values compatible with the current selection; that way, a selection is a facet and the highlighted items are the results of the facets. This system, however, was not built with large-scale datasets in mind; large-scale datasets will have millions of items per category and loading nodes with that many items (and consequently, facets) would be ineffective.

Facete is an application for faceted search over RDF with geographical data; it requires the dataset be loaded in a SPARQL endpoint. Facete works in three views: the first view shows all properties of the dataset, where selecting a property will display possible values for the facet; the second view displays the items that match the query; and the third view displays the item's location on a map. The system needs to (or at least tries to) relate each item with geographical coordinates, so many types of items are suitable for this system. We provide an example screenshot of Facete in Figure 2.4.

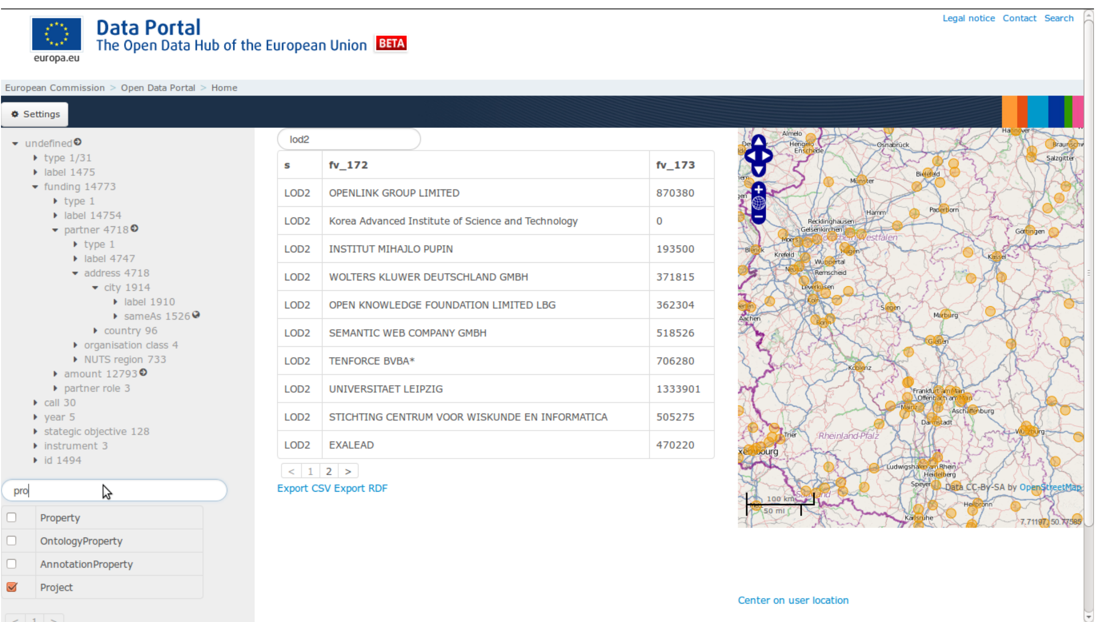

Figure 2.4: Facete interface

SemFacet is a faceted browsing system that requires a SPARQL endpoint and an RDF dataset with its ontology as input. The initial search is performed using a keyword over the meta-data of the items, using information retrieval techniques; relevant facets are computed

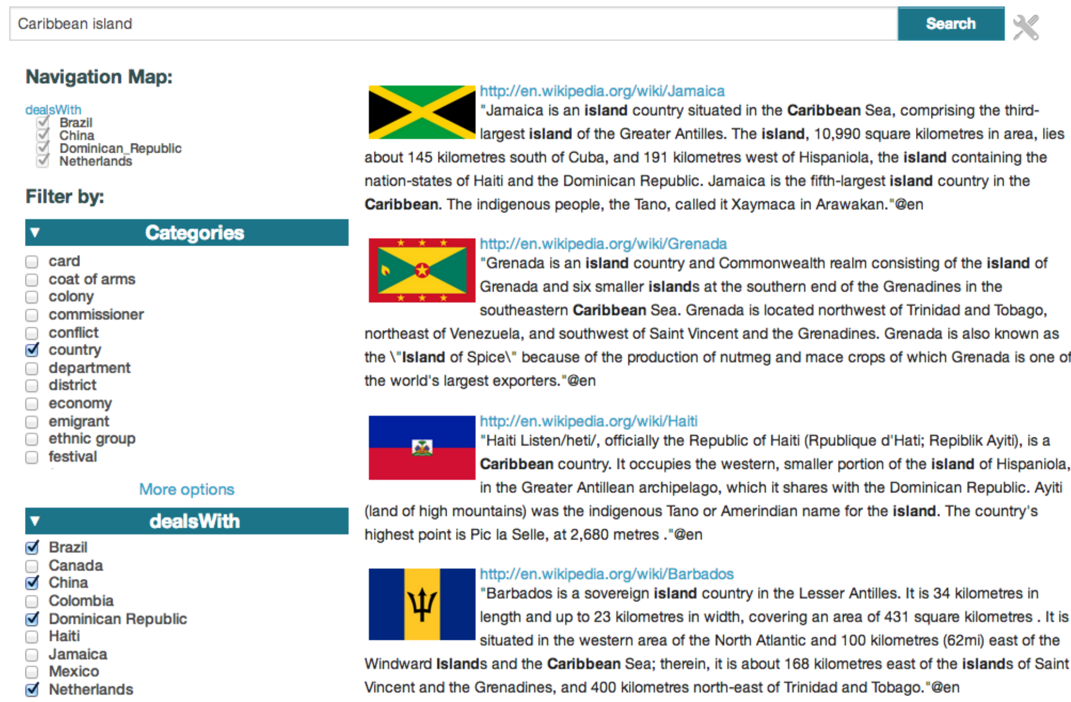

Figure 2.5: SemFacet interface

using the ontology file of the dataset. Facets and values are displayed at the left of the screen, while on the right side the results with an image and description are shown. While SemFacet can use the input ontology to perform reasoning about which are the relevant facets, this limits the scalability of the system, where evaluation reported in the paper deals with only a subset of the YAGO dataset. We tried loading Wikidata into this system, but it does not work, not displaying any results or facets after one day of processing the dataset. We provide an example screenshot of Facete in Figure 2.5.

eLinda [10] is a browsing interface that shows in a bar chart the possible types or properties of the entities of the dataset. It requires an ontology for the dataset and class type hierarchy; that way, it can make a bar chart of the subtypes, allowing the user to select a subtype to refine the current results. Alternatively, it is possible to make the chart based on the properties, similar to facets. The main difference between this system and a standard faceted browsing interface is eLinda does not show the resulting entities, instead it groups them based on its type or properties to select the next group; however, these groups are a way to suggest "facets" to the user. This system uses a SPARQL endpoint to perform the queries and assumes an ontology to create a class hierarchy.

2.2 Information Retrieval

Information Retrieval is an area concerned with finding information resources relevant to a user need from a large collection of resources. Searches can be directly over text, metadata or both. In this work, an information resource is an entity identified by an IRI; this resource contains text on the labels, alt labels and description, and its metadata are the pairs of properties and values associated with it.

2.2.1 Inverted Index

To do searches over these information resources, they need to be indexed in some form. Information Retrieval call these indexes *inverted indexes* (since it inversely maps keywords to documents).

Inverted indexes store in a virtual document the information resource identifier, its contents in fields, each field for a different type of content; for example, for the Wikidata dataset, label, alt label and description are in separate fields. The contents of the documents are processed at the moment of index creation to allow fast searches.

- *Text fields* contain text and ignore stop words and relate similar words.
- *Numeric fields* can contain any numeric format (integer, double) and provide support for range queries and sorting techniques.
- *String fields* look for exact matches character by character
- *Stored fields* are not for querying but their contents are saved in the document.

Searches are done over the document's fields; documents that fit the querying criteria are returned, but the order of the results is also important. In the case of text fields, different fields may be more or less important; for example, a matching word in the label field is more important than in the description field, as well as the frequency of the matching word (more frequency means a higher relevance) and how rare is the word (rarer words are more important). A popular ranking measure used in practice is known as TF-IDF (term frequency, inverse document frequency) and helps to determine what documents are more relevant; results are then sorted accordingly.

2.2.2 Ranking Importance

Other measures help to determine the importance of a document independent of a query; for example that Earth, the planet, is more important than the element. Some measures consider links between documents: a document with more in-links is more important, but

not every in-link is equally important; links from a document with fewer out-links are more important and links from an important document are more important. One measure that implements this idea is PageRank [4]. But, there are some scenarios to consider: the first is when a document has no links and the second is when two documents link only to one another. In order to avoid high ranks from these cases, there is a probability to not follow any link and jump to a random document.

The process starts all documents with the same rank such that the sum of all document's rank is 1. Then, the rank of each document is shared to the documents it links, so a higher-ranked document give higher rank to its links, while documents with lot of links give a low rank because its rank is divided across all its links. With the probability to jump randomly (usually set as 0.15), this process is recursive and guarantees to converge, so after a number of iterations each document's rank is fixed.

PageRank can also be used to rank nodes in an RDF graph where each resource `A` contains a link to another resource `B` when there exists a triple `(A, p, B)` for some property `p`. Combining the ranking from the PageRank algorithm and the relevance given by the TF-IDF measure, documents in a query are returned in that order, so the user can always see first the most important results (as estimated by the system) from a large document collection.

Chapter 3

System Overview

The system aims to support faceted browsing over the Wikidata dataset. First, the user needs to start with a basic search before adding facets to the results. A good initial search can be a keyword or a type, so the system will allow both methods. With the results of the initial search displayed, the user needs a list of all the possible facets in order to select one. With a new facet selected, a new results set will be displayed and the user continues until he or she has found the resources that match the required facets.

Example 3.1. A user selects the type person, but the results have more than three million people, so to refine the search the user adds the facet *gender: male*. The user keep adding facets, like *occupation: sportsperson* and *country: US*, to see all people who satisfy the criteria.

Example 3.2. Another browsing session starts with a user searching for "john" as keyword; the results contain resources with the keyword in their text fields. The user starts to add facets to filter the results; this time, the type is just another facet, so for example, *type: building* is added to the results. This way, a user can use a keyword and a type in a browsing session and can add more facets as desired.

3.1 User Interactions

In order to fulfill the previous description, the system has four major user interaction stages that connect to each other powering the faceted browsing interface. These interactions will be called *Keyword*, *Type*, *Results* and *Facets* based on their main function. In summary, the *Keyword Interaction* allows to perform a search over the system using a keyword, while the *Type Interaction* provides the user all possible types in the dataset to select from. Once the user has selected a type or specified a keyword to search over the resources, the query is sent to the *Results Interaction*; this stage recovers all of the relevant data for every resource that matches the query including all properties that they have. In the Results Interaction, it is

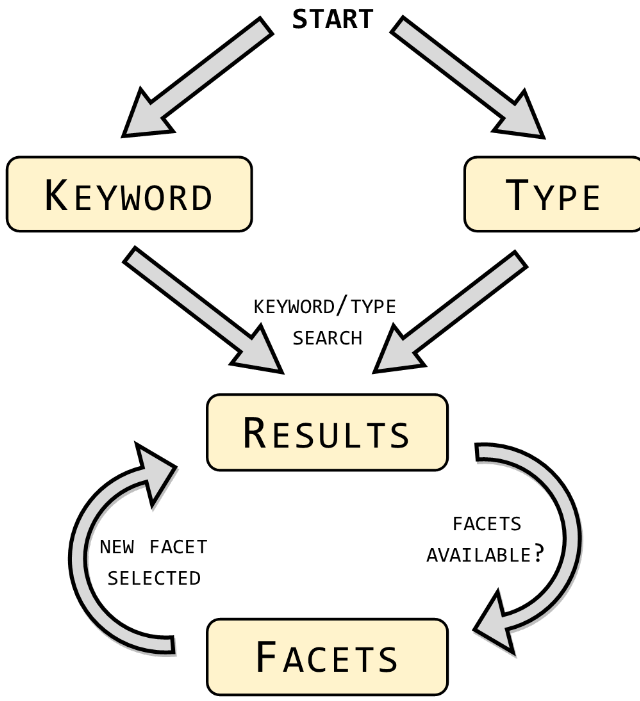

Figure 3.1: Flow of user interactions

possible to select a property to send to the *Facets Interaction*, obtaining all possible values given the previous results and the property selected; this way, all possible values for a given property are available to make a new query and continue with these last steps. Other than for the Keyword Interaction, a main design goal is that the user is only offered interactions that will lead to non-empty results; we will discuss how this goal is accomplished in future sections.

Example 3.3. For Example 3.1, the user interacts with Type to select *people* from the list of types and starts the browsing session. In Results, the user sees the resources with *type: person* and their other properties; in this case, selecting *gender* takes the user to the Facets Interaction. Inside Facets, a list of all values for *gender* are displayed; the user selects *male* to return to the Results Interaction, but now, the results include the facet *gender: male* as well the previous *type: person*. The user can continue selecting properties from Results and values in Facets. For Example 3.2, the user starts writing a keyword in the Keyword Interaction; the Results and Facets interactions then work the same way as if the session started with a type facet.

Next, each one of these interactions will be explained with more detail. Example screenshots of the user interface are provided in Chapter 5, which discusses the front end.

3.1.1 Keyword

The Keyword interaction is one of the first with which the user can start the browsing session. Because it is the starting interaction, it does not require any previous data nor server request. Here, the user writes a keyword to search in the labels and description of each resource; then the query will be executed and the results will be sent to the Results interaction; the Keyword interaction then finishes.

3.1.2 Type

The Type interaction is the other way to start a browsing session: the user enters a text (the name or the first letters of a type) to search what types match that text. Then, using the input text, this interaction looks for all the types registered to obtain a list of possible types; such a list is displayed by autocomplete to the user, where he or she then selects one and sends the respective facet to the Results interaction. Matches are computed based on autocomplete for types whose labels match the current user input. This step is necessary because it may not be obvious what is a type or not, so to make sure a user enters a valid type in the first place, it is required to select one from the list given based on matching as a prefix what he or she typed.

Example 3.4. For Example 3.1, the user starts typing “person” in the input box; the list will provide the user all types containing the letters as a prefix in their label or alt label field. When the input has “p”, the list displays as auto-complete the types: **p**erson, **p**aper, web **p**age, etc.; with “per”, the options would be: **per**son, **per**iodical, **per**sonality and so forth.

In order to implement this, there is an index with a document for every type containing its identifier (because we are working on Wikidata, the prefix is dropped, and only the Q number is stored), its labels, the number of resources that have this type or its frequency, and a ranking that determine its relevance; the index is further explained in Section 4.3. To obtain the list of types, a query over this index is run looking for matching labels (query structures are explained in Section 4.4). The first fifteen results are returned sorted by the ranking; the number of results is also returned such that the user knows how common that type is. It is possible to use the number of results as a measure to determine if the type is correct or not because the labels can and will be ambiguous, so this measure is a way to overcome this issue, and avoid the user selecting an incorrect type.

3.1.3 Results

The Results interaction is the main part of the system: this interaction receives a query (keyword and/or facets) and outputs the resources that match the criteria, providing its name, description, image (if available) as well as all possible properties for all results.

To allow this interaction to return all this information, there is an index with a document per resource containing the following data: identifier (again storing only the Q number), labels (names the resource is known for), description, type (its Q value), image, ranking, properties that the resource has, and all pairs of properties and values. Full details of this index are specified in Section 4.1.1.

To process a keyword query, the system looks in the labels and description for the keywords. If it is a type query, it is necessary only to check the type field of the document and return the matching documents. The Results interaction will be called again after a query if the user has selected a new facet for the previous results set. In this case, for each facet a new query is required to run over the index looking for documents with the specified pairs of property and value. All queries need to join with an AND operator because the results set needs to satisfy all criteria.

For all documents obtained from the search, we need to check all the properties associated with any current result and add them to a list to allow the user to select a new property as a facet; the values of the properties are loaded lazily to not overload this interaction with work; the values for a property will be computed in the next interaction when a particular property is selected. The fifty most relevant results based on the rank value of the documents are displayed with its labels, description and images. In summary, the Results interaction runs the specified query and returns a list of all properties that the entire results set has and an overview for the first fifty results.

Example 3.5. For Example 3.1, the Results interaction displays the resources of type person, such as *Jimmy Wales, Nelson Mandela, Queen Elizabeth* and also the properties of the resources, like *gender, country, occupation, award received*, etc.

3.1.4 Facets

The Facets interaction receives as input the current query from the Results interaction and a property available from the results itself. Using this data, the component runs the same query as before but adds the condition that results have the selected property. From the resulting documents, each possible value of the property is recorded and then returned to the user to select a value for the property and obtain a new set of results with the new facet added.

This approach is potentially costly, especially if there are many results; however, it guarantees that no query will return empty results. Avoiding empty results is a very important feature in this work and it makes a major difference between this system and writing a query in SPARQL because in the latter there is no way of knowing what properties or values each resource has, whereas in this system, everything is computed before showing it to the users;

this idea reinforces the principle of querying RDF with no knowledge of the structure of the dataset.

Example 3.6. Referring back again to Example 3.1, after the user selects the *gender* property; the Facets interaction will display the list of possible values for the property including *male, female, transgender*; the list will not display invalid values such as *US, politician, queen*. In the second facet selection of the example, when the user selects the *occupation* property, valid values for the property but not for the current query are not displayed; for example, *queen* is a valid value for *occupation*, but the current query contains the facet *gender: male*, so the facet *occupation: queen* would produce empty results so it is not available in the list of values.

A cache is necessary to save all values and all properties if the results set is too large; this way computing facets will not take minutes for each property. The cache is a document for each entry cached using as identifier the type and facets selected; when a query is run on the system it checks if there are cached results present for that query; if so, the cache returns the data stored; if not, the values for the property are computed from scratch. Creating a cache for all possible combinations of types and facets with many results can take days and the time may increase to over a week with the new triples added to Wikidata. It is important to note that it is impossible to cache over keyword searches; in that case if the results set is too large, calculating the facets may take a couple of minutes. Cache conditions and structure will be explained in further detail in Section 4.5.

3.2 Query Expressiveness

3.2.1 Operations

In order to define the operations that the faceted interface allows, we need to define these operations with respect to an RDF graph.

Notation: The RDF graph G contains a set of triples (s, p, o), where s is the subject, p the predicate and o the object. $\pi_S(G) = \{s \mid \exists p, o : (s, p, o) \in G\}$ is the set of all subjects of G, similarly $\pi_P(G)$ and $\pi_O(G)$ are the sets of all predicates and objects respectively. The elements of $\pi_S(G)$ which are IRIs are known as *entities*; the elements of $\pi_P(G)$ are the *properties*. For a given entity s and property p, any o such that $(s, p, o) \in G$ is the *value* of the property p for entity s.

Keyword search: Entities with values for label and/or description properties (`rdfs:label`, `skos:altLabel`, `schema:description`) can be searched with a keyword. The keyword search function is defined as follows: $\kappa : \mathbb{S} \to 2^{\pi_S(G)}$, where $\mathbb{S}$ is the set of strings. The function will return a set of entities whose label/description values match the keyword.

Type selection: Another way to start a session is by selecting a type. The *type* of an entity s is the value for the property `wdt:P31`. The set of types in a graph G is denoted as $T(G) := \{o \mid \exists s, p : (s, p, o) \in G \text{ and } p = \texttt{wdt:P31}\ \}$. Then, the type selection function is defined as: $\tau : T(G) \rightarrow 2^{\pi_S(G)}$, where $\tau(t) := \{s \mid (s, \texttt{wdt:P31}, t) \in G\}$. Thus, the function will return all entities whose type is t.

Facet selection: For a results set, the user can select facets to refine the current results. A *facet* is a property-value pair that at least one entity in the current results set has. For a given current results set $E \subseteq \pi_S(G)$, the projection from G of all triples with a subject term in E is denoted as $E(G) := \{(s, p, o) \in G \mid s \in E\}$. Then, the facet selection function is defined as: $\zeta : 2^{\pi_S(G)} \times \pi_P(G) \times \pi_O(G) \rightarrow 2^{\pi_S(G)}$, where $\zeta(E, p, o) := \{s \mid (s, p, o) \in E(G)\}$. This function returns a new result set which is a subset of E and whose elements contain the specified property-value pair.

Facet navigation: A navigation session consists of sequential facet selections of a set of entities which started with a keyword or type selection.

– $\zeta(\zeta(...(\zeta(\kappa(q), p_1, o_1)..., p_{n-1}, o_{n-1}), p_n, o_n)$
– $\zeta(\zeta(...(\zeta(\tau(t), p_1, o_1)..., p_{n-1}, o_{n-1}), p_n, o_n)$

Note that the facet selection function is commutative and that the order of the selection is not relevant for the final results set. Hence, the notation can be simplified as a conjunction of criteria with a keyword or type as starting point.

– $\kappa(q)\ [\zeta(p_1, o_1) \wedge ... \wedge \zeta(p_n, o_n)]$
– $\tau(t)\ [\zeta(p_1, o_1) \wedge ... \wedge \zeta(p_n, o_n)]$

3.2.2 Analysis versus SPARQL Queries

In the previous section, the main operations of the system were specified. SPARQL is the current standard to query RDF triples, so now we discuss how these operations can be expressed as a query and what is the best way to implement such operations in the system we want to develop.

The keyword search operation is not possible to do efficiently in SPARQL: SPARQL only supports basic string filters, so efficient keyword search must be supported using external indexes. Type selection and facet selection operations are possible to translate to a SPARQL query specifying the matching triples in the query clause. In summary, faceted navigation is possible in SPARQL if it starts with a type selection; the SPARQL query will look like the following:

```
SELECT DISTINCT ?s ?p ?o
WHERE {
  ?s wdt:P31 t .
  ?s p1 o1 .
```

```
  ...
  ?s pn on .
  ?s ?p ?o .
}
```

Listing 3.1: Facet navigation on SPARQL

However, the resulting triples need to be filtered by entity to display the labels, images and descriptions. After that, we need to filter by properties and values to obtained the possible facets of the current result set. This approach is costly due to the need to process and order multiple times a potentially long list of triples.

Due to the need of an inverted index to effectively support keyword searches, it is possible to add to the documents in the index all the necessary information of a entity including properties and values. In that case, all the necessary information needed for a result set including labels, descriptions, images, property and values is within the respective documents of the entities matching the criteria. The latter approach is faster than performing SPARQL queries and filtering the triples afterwards. For a system with fast user interactions as a main objective, performance is a key issue, and hence in this work we prefer custom inverted indexes rather than using more general SPARQL indexing and optimization methods.

Chapter 4

Back end

In the previous chapter, the user's interactions with the system were explained with the usage of indexes to access the data. In this chapter, all steps from the raw data to the indexes and back-end queries will be explained.

All the back end code is written in Java. This work uses Apache Lucene[1] as an inverted index platform. All indexes and queries are done with Lucene. RDF parsing is done using the RDF4J[2] library.

4.1 Initial index

The raw data is obtained directly from Wikidata. Wikidata provides full RDF dumps in different formats. This system uses the "truthy" dumps: they use the NT format and contain only direct statements, which means they do not include meta data like references. The dump is also grouped by subject; this property of the dump is very important and it is used to build the index.

4.1.1 Structure of the documents

Each document in this first index contains the following fields:

- **SUBJECT:** String field. The IRI of the subject or resource without the prefix. For Wikidata, this identifier is a number with the letter `P` (for properties) or `Q` (for entities and types) at the beginning
- **LABEL:** Text field. Value that represents a name for the resource (that may be ambiguous). In the dump, the label is the value of the triples with the predicate `http://www.w3.org/2000/01/rdf-schema#label`. These values have a language tag.

[1] http://lucene.apache.org/
[2] http://rdf4j.org/

For each language, a different field in the document has to be created, for example, **LABEL-en** and **LABEL-es**.

- **ALT LABEL:** Text field. Other names the resource is known by. The predicate for these values in the dump is `http://www.w3.org/2004/02/skos/core#altLabel`. Different fields must be stored for each language. This field can store multiple values for the same document.

- **DESCRIPTION:** Text field. Small description of the resource, which helps to disambiguate the resource from others with the same labels. Triples with a description have the predicate `http://schema.org/description`. Descriptions have to be stored for each language as per the previous two fields.

- **IMAGE:** Stored field. URL of an image that represents the resource. The predicate used in the triple is a Wikidata property with the corresponding prefix and the number `P18`.

- **TYPE:** String field. IRI of the resource that represents the type of this resource. All types are a Wikidata IRI starting with the letter `Q`. The predicate in such triples is the Wikidata property `P31`.

- **PROPERTY:** String field. All Wikidata properties that this resource has. This field can store multiple values. All values in this field starts with the letter `P` (as the Wikidata prefix is always omitted).

- **PO:** Predicate-Object, string field. All pairs of Wikidata properties and Wikidata resources that this resource has with the string `##` as separator. In other words, this field stores the triples where the subject is the resource of the document, the predicate is a Wikidata property (starting with the letter P) and the object is another Wikidata resource (starting with the letter Q). This field can store multiple values per document.

Note that aside from labels and descriptions, we do not store datatype values in the index since the current version of the browser does not support range queries.

Example 4.1. Using as an example the Wikidata resource in Figure 2.3, its respective document in the index is as follows:

- **SUBJECT:** `Q42`

- **LABEL-en:** `Douglas Adams`

- **ALT LABEL-en:** `Douglas Noël Adams`
`Douglas Noel Adams`

- **DESCRIPTION-en:** `English writer and humorist`
- **IMAGE:** `commons.wikimedia.org/wiki/File:Douglas_adams_portrait.jpg`
- **TYPE:** `Q5` *(human)*
- **PROPERTY:** `P31` *(instance of)*
  `P69` *(educated at)*
  ...
- **PO:** `P31##Q5` *(instance of: human)*
  `P69##Q691283` *(educated at: St John's College)*
  `P69##Q4961791` *(educated at: Brentwood School)*
  ...

The text in italics is only an explanation of the actual value and is not part of the index. The PROPERTY and PO fields contain more values than listed in the example for the actual document `Q42`.

4.1.2 Building the index

To build the index, the dump is read line by line. Each line contains a triple and triples are grouped by subject. All triples with the same subject are processed; when a new subject is read, all data of the previous subject is written in a document and the process starts over with the new document for the new subject.

For each subject, a new document is created with its IRI in the SUBJECT field. The process then checks the predicate position of the triple; if it is the predicate for label, alt label or description, the value is added to the document with the respective language tag (if that language is supported). If the predicate is a Wikidata property, then the process checks the object in order to see if it is a Wikidata resource. In that case, the property is added to a list of all distinct properties and the pair of property and value is added to the document. Only when the program reads a new subject is the list of all properties added to the document in order to avoid duplicate entries of the same value.

```
document ← new Document()
for each line in dump:
  s, p, o ← line
  if s != previousSubject then
    save(document)
    document ← new Document()
    addTo(document, field('SUBJECT', s))
  if p = 'rdfs:label' then
```

```
    addTo(document, field('LABEL', o))
  else if p = 'skos:altLabel' then
    addTo(document, field('ALT_LABEL', o))
  else if p = 'schema:description' then
    addTo(document, field('DESCRIPTION', o))
  if p.startsWith('wdt:') then
    if p = 'wdt:P31' then
      addTo(document, field('TYPE', o))
    if p = 'wdt:P18' then
      addTo(document, field('IMAGE', o))
    if o.startsWith('wd:') then
      addTo(document, field('PROPERTY', p))
      addTo(document, field('PO', p + '##' + o))
save(document)
```

Listing 4.1: Pseudo code of the algorithm

4.2 Ranking data

Once the index is already built, it is possible to do some queries but first it is necessary to determine what resources are more important than others. A well-known algorithm to rank documents is PageRank (see Section 2.2.2).

As explained before, an RDF dataset can be interpreted as a graph. Since each document in the index has the PO field that links to other documents, we can build a directed graph from the index directly. With this information, a ranking is calculated for each document. This is equivalent to applying PageRank over an RDF graph considering directed links from subject to object; the predicate (edge label) is not considered. This is similar to applying PageRank over the webpages of Wikidata, where the page of an entity A contains links to the page of an entity B if, in the RDF graph, there is a triple (A, p, B), for some property p.

With the ranks computed, a new index is built with the same structure as before but with two new fields.

- **RANK:** Numeric field. This is the value obtained from the PageRank algorithm over the RDF graph built from the first index.
- **RANK STORED:** Stored field. Lucene does not allow to obtain values of a numeric field; because of that, this field stores the rank value so it can be recovered as per the other fields.

Together with the document and ranking metadata, this index contains the most important data from the RDF dump, so this index will be known as the **Data Index**. Most operations and queries will be done over this index.

4.3 Types index

The Types index stores each type present in the Data Index with its label and aliases, in order to allow searches by keyword over all types obtaining its respective `Q` value. This index is built taking the data directly from the Data Index. Hence, this index allows for searching over only types (e.g. person, country, etc.) and has type-specific information not in the Data Index; we mark these novel fields with *.

Documents in this index have the following fields:

- **ID:** String field. The `Q` value of the type.
- **LABEL:** Text field. Label of the type; same content as the Data Index.
- **ALT LABEL:** Text field. Alt labels of the type; same content as the Data Index.
- **FREQUENCY*:** Numeric field. Number of resources from the Data Index that belong to this type.
- **FREQUENCY STORED*:** Stored field. Same number as the FREQUENCY field but stored, in order to be recovered when the document is queried.
- **RANK:** Numeric field. Ranking of the resource, previously calculated and stored in the Data Index.

Example 4.2. The document for the type *human* is as following:

- **ID:** `Q5`
- **LABEL-en:** `human`
- **ALT LABEL-en:** `person` - `people` - `humankind` - `human being`
- **FREQUENCY:** `3595226`
- **FREQUENCY STORED:** `3595226`
- **RANK:** `0.00517`

4.4 Queries and results

To query over the Data Index and the Types Index, there are four main types of queries: by identifier, by keyword, by type and by property-value pairs; the last two types only apply to the Data Index. It is also possible to do mixed queries.

4.4.1 Queries by identifier

Queries by identifier are the most simple type of queries. The query needs a `Q` value as input to search documents where its SUBJECT (for the Data Index) or ID field (for the Types Index) is the given input. These queries only return a single document because the field stores the IRI of the resource, which is unique to that document and resource.

4.4.2 Queries by keyword

These queries use the fields LABEL, ALT LABEL and DESCRIPTION (only in the Data Index) from the document; because these are text fields, the terms in the query need to be processed as well. Such a query receives a keyword; then the keyword is processed and the index is used to look for matches in the words previously processed in the fields of the document.

Because the result is typically more relevant if the keyword is in the LABEL field than if it is in the DESCRIPTION field, the results should not be sorted directly by the rank, so the rank is added to the scoring function of the query by multiplying the score from Lucene with the ranking.

Example 4.3. For the keyword query using "adam" over the Data Index, a multi-label query is performed with different weights for each label. The query is done with the following weights: `{LABEL-en: 5; ALT LABEL-en: 2; DESCRIPTION-en: 1}` and search for `{LABEL-en: adam; ALT LABEL-en: adam; DESCRIPTION-en: adam}`. The results contain resources with the keyword in these fields, such as: *Douglas Adams*, *Adam* (given name), *Adams* (surname), *Adams County* (one resource for every county in different states).

Example 4.4. Performing a keyword query on the Types Index is similar to the previous example, but because there is no description field, weights for fields are dropped. However, a prefix query is used to enable the autocomplete function described in the previous chapter. The full query is `{LABEL-en: hum; ALT LABEL-en: hum; LABEL-en: hum*; ALT LABEL-en: hum*}` and it will return the types *human, human settlement, fictional human*, etc.

Keyword queries are specific to the language in which the query was made. The same keyword will likely return a different results set depending on the language used. Also, stop words and word normalization (stemming, lemmatization) work differently for each language.

4.4.3 Queries by type

Queries by type return the documents whose TYPE value is that specified by the user. Because all types stored in the document are `Q` values, in order to make a query by type the

`Q` value of the type must be known (i.e., selected by the user through autocomplete).

Example 4.5. The query that returns all documents of type *human* (`Q5`) is `{TYPE: Q5}`.

This kind of query is a good starting point for a faceted browsing system: since all documents of the same type share certain properties, it is easier to find relevant facets for the current results set.

4.4.4 Queries by property-value

Similar to queries by type, these queries use the PO field from the document. There is no language issue because the pair stored in the PO field uses the `P` values with a `Q` value. These values need to be known beforehand; however, again, the interface will help users select the `P` and `Q` values required by providing autocomplete interactions.

Example 4.6. The query that returns all documents that have the property *gender* (`P21`) with the value *male* (`Q6581097`) is `{PO: P21##Q6581097}`.

Note that such queries correspond to adding facets to the current query.

4.4.5 Mixed queries

A mixed query is built from "atomic" queries joined with the logical operator AND or the operator OR. Such queries are useful to do more complex faceted browsing that combines criteria; the initial query is either a query by name or by type and then the facets are one or more queries by property-value; these query criteria are combined with a conjunction using the operator AND.

Example 4.7. The query that combines all previous examples (*adam* keyword, *type: human* and *gender: male*) is `{LABEL-en: adam; ALT LABEL-en: adam; DESCRIPTION-en: adam; TYPE: Q5; PO: P21##Q6581097}`.

In summary, the Data Index and these queries are enough to power an initial faceted browsing interface. However, there still some issues to address, such as that the queries by type and property-value require `P` and `Q` values and the users who will use the system do not know what resource represents a certain `P` or `Q` value. To solve this, queries by identifier are used to quickly translate an IRI to the LABEL field. More details will be provided in the next chapter, which describes the user interface. Another issue is loading all possible facets with interactive runtimes when there are potentially millions of results; this challenge we now address.

4.5 Improving times and caching

With a results set obtained from a query, it is necessary to compute all properties and values to give the user possible facets for the displayed results. To do this, for each document in the results set, a list of all properties needs to be stored, translating the `P` value to its respective label and returning it to the user. When a `P` value is selected by the user, the same process is repeated for all possible values for the property selected.

Example 4.8. The results set for the query *type: human* contains about 3.6 million documents. These 3.6 million documents are read sequentially, in order to create a list of every value stored in their PROPERTY fields, where the resulting list contains over 350 unique properties; then, 350 queries by identifier are executed to translate the `P` values to labels, reading about 350 documents more. If the user selected the property *occupation*, which is defined for about 3.4 million of the results, such documents are read once more to store the values of their PO field starting with the `P` value of occupation; these latter values, which are about 7,800, need to be translated, thus making 7,800 queries and 7,800 more documents read. These two processes, which read approximately 3.5 million documents and involve 8,150 queries, take around two and a half minutes to complete.

IRI	Label	Documents	Properties	Time (s)
Q13442814	scientific article	6621865	50	404
Q4167836	Wikimedia category	4043768	96	230
Q5	human	3595226	358	135
Q16521	taxon	2270273	60	68
Q4167410	Wikimedia disambiguation page	1223596	147	34
Q11266439	Wikimedia template	857358	62	28
Q13100073	village-level division in China	588485	7	20
Q8502	mountain	476518	72	16
Q486972	human settlement	410027	126	31
Q532	village	293016	93	18

Table 4.1: Times of processing results for most common types

Queries that return a lot of documents thus take a lot of time in being processed (making lists of properties/values and translating them); more time than a common user would like to wait (see Table 4.1 for different times based on the number of documents and properties). To overcome this issue, a cache will be created with facets for certain queries already computed. Note that in order to keep the cache non-language-specific, the cache will store only the P values; thus, a subject query will be needed to translate the IRI to a label. A second cache index will also be needed to store the values that a certain property can take for a given

results set. This cache, however, does store the labels for every language supported because the number of values will be more than the number of properties. To sum up, the system will use two separate caches. The first links a results set with all possible properties and the second relates a results set and a property of the first cache to possible values of that property.

While we aim to have good times for computing the results, on the other hand, it is not reasonable to cache every possible query; hence we have a trade-off between time and space. It is important to note that there should be a limited number of properties that return a large result set; for example, the query *type: human* return more than 3.5 millions results, adding the facet *gender: male* reduce the results to about 2.8 millions; but, adding the facet *father: Barack Obama* reduce the results to only 2. Thus, the solution is determine which are the properties and values (and combinations) that produce lots of results to cache the corresponding queries; but, how much is "lots of results"? In order to calculate this threshold, for every property `p` the following values were calculated in a table:

- **Number of subjects:** How many resources have the property `p` in their PROPERTY field.
- **Number of values:** How many different values this property can have; this means, counting all the different pairs of property and values in which the property is `p`.
- **Frequency of the most common value:** How many resources would the query with the most common value for property `p` return.

Example 4.9. These are the values for the property *gender*.

- **Number of subjects:** 3,371,738 (documents with the property)
- **Number of values:** 15 (including: male, female, transgender, intersex)
- **Frequency of the most common value:** 2,793,564 (documents that have the most common value for this property, which is male)

With this information, the **frequency** is the maximum number of documents produced by a query containing the property `p` and the number of **values** is how many additional queries and document readings are needed when `p` is selected for a facet; therefore, if the values of `p` are not cached, **number of values** queries need to be executed if `p` is selected.

If we sum the **number of values** across all properties, the resulting value is the total of work needed for computing the values for every property. Now, if we exclude the properties whose **frequency** is less than a certain value (a **threshold**), the sum would represent the extra work but only for included properties, which are the ones that produce more than

threshold results. That means, the greater the **threshold**, the fewer properties will be included in the sum.

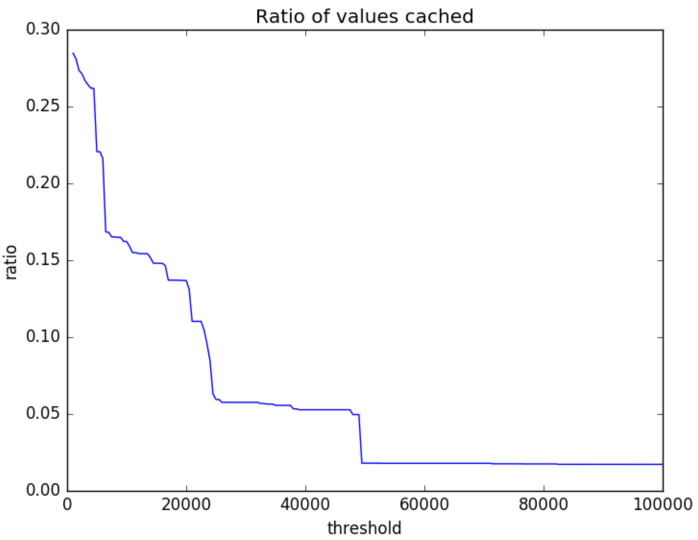

Figure 4.1: Ratio of values that need cache versus threshold

Figure 4.1 plots the ratio (current sum of **number of values** divided by the total sum) for different **thresholds**. As expected, the figure shows that starting from 50,000, the ratio stays the same; this means that caching properties whose **frequency of the most common value** is above 50,000 will only require caching 0.018 of queries, while only requiring live computation of results sets with fewer than 50,000 entities. Thus, the threshold we select is 50,000; queries that contain more than 50,000 results need to be cached. A threshold greater than 50,000 will not make much difference, because, the ratio would not drop (see Figure 4.1); with a threshold less than 50,000, the cache would include (much) more data, increasing the cost of creating such a cache. To compute the facets for 42,632 results without a cache takes 1 second, which we deem acceptable.

In order to determine which queries will return more than 50,000 results, the RDF dump is read again doing the following: For each subject, a list is made with its type and all combinations of property-value pairs.

Example 4.10. Assume a subject `Q1`, whose type is `Q2` and whose pairs are `P1##Q3`, `P2##Q4` and `P3#Q5` in the PO field; then, for this subject this list is created is: `{Q2, Q2||P1##Q3, Q2||P2##Q4, Q2||P3##Q5, Q2||P1##Q3||P2##Q4, Q2||P1##Q3||P3##Q5, Q2||P2##Q4||P3##Q5, Q2||P1##Q3||P2##Q4||P3##Q5}` . This list contains all possible non-keyword queries (which starts with a type and then facets) that can have this subject as a result. Clearly this list can be exponential in the number of pairs that an entity has; however, resources do not have

that many properties for this to become an issue; the computation of the list for all entities takes around 4.5 hours (full details in Table 6.1).

The lists for every subject are combined and the elements counted, resulting in a map where the key is a query and the value is the frequency of the key in the aggregated list; in other words, the value is the number of results that the query would return. As mentioned earlier, we were looking for queries with more than 50,000 results which we can determine from this analysis and add to the **need-caching** list. In the current version of the system, the number of queries that fit the criteria are 141; that means that the cache needs to store the properties and values for 141 different possible queries.

4.6 Cache indexes

In the previous step we identified the keys of queries that require caching to maintain interactive runtimes. We now describe the indexes used to cache both the possible properties for every such query and the values for each of the previous properties.

4.6.1 Cache for properties

The cache for properties consists of an index which maps a query key to its list of available properties for selecting a facet. The cache is built by executing every query in the **need-caching** list and saving the properties contained in the results. Documents in this index have the following fields:

- **ID:** String field that contains a query key from the need-caching list.
- **PROPERTY:** Stored field consisting of all properties associated to the query key of the ID field. Only `P` values are stored here.

Only identifier queries are needed over this index, in order to obtain the stored properties when a user query matches one of the query keys stored in the cache. If the user query does not match any document, properties are calculated from scratch using the results. Nevertheless, queries for translating `P` values are still needed, since this cache does not store the labels for the properties.

4.6.2 Cache for values

Similarly as before, the cache for values is an index that maps an identifier to a set of values. This index is built using the cache for properties as a base: for every pair of query key and property, its values need to be stored in a cache. Documents in this index have two fields:

- **BASE:** String field that identifies the value cached; the identifier is a query key and one of its properties using the same separator ||. For example, `A||P1` means that this document stores the cache for the property `P1` in the query key `A`, (given results for query `A`, what values are possible for property `P1`).

- **VALUES:** Stored field that contains the available values associated to the identifier on the BASE field. This field stores the IRI (`Q` value) and its label for every language supported.

When the user selects one property of a cached query, the values are loaded from the cache using a query by identifier to this index. Unlike the cache for properties, this cache stores the labels so no queries for translating `Q` values are needed; the data stored in the VALUES field is sent directly to the user interface.

Chapter 5

Front end

The front end is where the user interacts with the system; it also provides the interface that connects the user with the indexes. In other words, the front end allows user-friendly interactions that will be translated into queries over the indexes described in the previous chapter.

5.1 Home page

The home page is where the user starts a faceted browsing session in one of two ways: keyword search over all documents or selecting a type. The home page displays the type search by default because it is the recommended way to start browsing.

Figure 5.1: Home page

In the type search, an input box is provided to write the first letters of a type, which will

be matched against the Types Index to offer autocomplete suggestions. After half a second of no input has passed (to not send too many autocomplete request as the user types), a request to the back end is sent to do a *query by keyword* over the Type Index (which searches prefixes, see Example 4.4); the results are returned to the front end recording the `Q` values (invisible to the user) and the label (or alias, depending on which matches the user's input) and number of resources of that type (shown to the user) sorted by descending ranking (based on PageRank); these values are already stored in the documents of the index. This process can be seen in Figure 5.2, which displays autocomplete suggestions based on the text in the input box; the types suggested also include the number of results that will be returned.

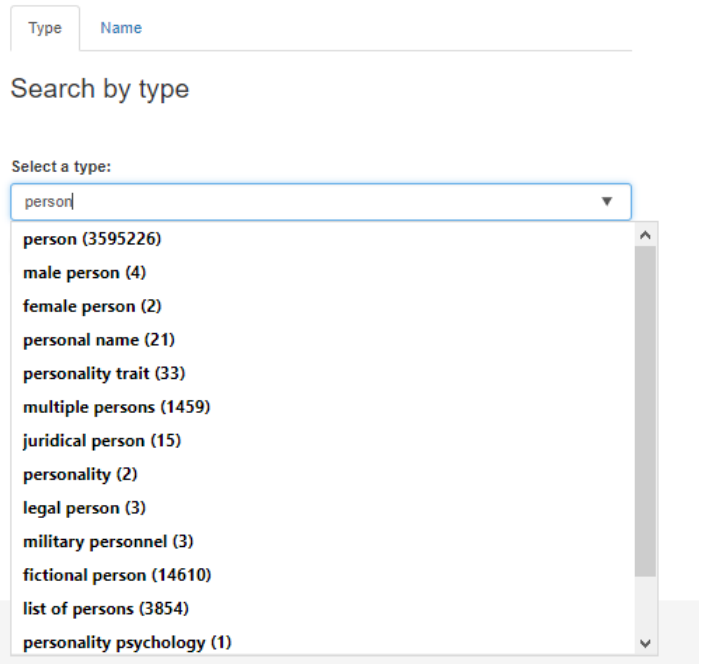

Figure 5.2: Type selection

Then the user must select one entry from the list of results; the user cannot submit anything other than a valid type from the list to continue; otherwise, the `Q` value would not be stored to perform a *query by type* over the Data Index as seen in Example 4.5. This user interaction corresponds to the Type Interaction, mentioned in Section 3.1.2.

Note that type searches on the input box use (partial) keywords; in that sense, the list of types may be different depending on the current language. The home page defaults to the English version, but a language selector is located in the top right corner to change the language at any time (seen in Figure 5.1).

To start a faceted browsing session using a keyword search, the tab to change to this method has to be selected. Once selected, the user can enter a keyword and start the search any moment; there is no list like the type search. The *query by keyword* function is

executed over the Data Index, which searches the text on the LABEL, ALT LABEL and DESCRIPTION field (as explained in Example 4.3). This user interaction corresponds to the Keyword Interaction (see Section 3.1.1).

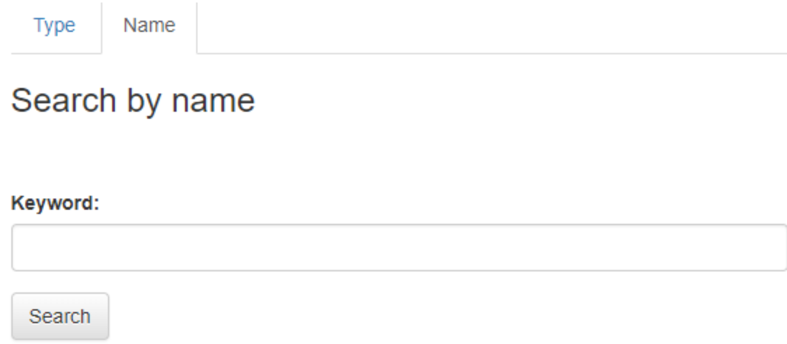

Figure 5.3: Search by keyword Tab

Whether a search by type or keyword was performed over the main Data Index, a results set of documents is obtained and then sent to the results page.

5.2 Results page

The results page contains the documents returned by the keyword, type or combined (faceted) query. This page is divided into two sections: the navigation bar at the left, which displays the current query and available properties, and the results entries at the right, displaying the metadata of the documents. An example is given in Figure 5.4.

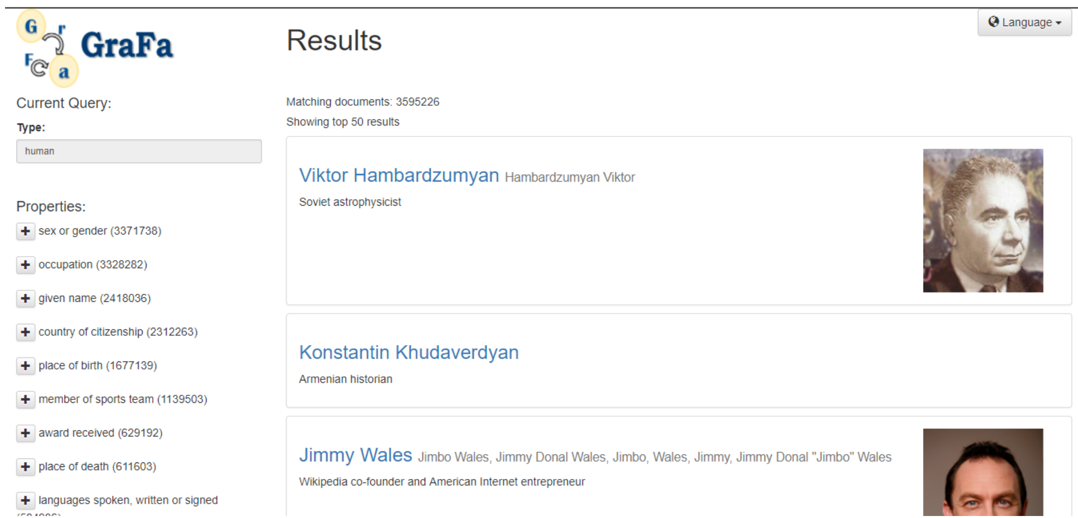

Figure 5.4: Results page

5.2.1 Navigation bar

The navigation bar displays the logo of the system (to return to the home page), the current query, and the properties with the number of documents that have some value for it.

The current query consists of a type or keyword and zero-or-more facets. The real query executed over the index requires the IRI of the type or facets; however, to keep the system user-friendly, the query on the navigation bar is displayed with the corresponding labels, performing additional *identifier queries* to translate the IRIs to labels. All facets in the current query section can be removed; however, the type or the keyword (in other words, the initial search) cannot; to start a new browsing session, it is necessary to return to the home page.

Below the current query, there is a list of all properties of the results set that do not have a facet already selected, which means that if the current query has a facet with property `P1`, this property will not be displayed in the list. The list is ordered by the number of resources that have that property. The list is manually created reading all documents from the results set if the current query is not cached; note that all queries using a keyword at the beginning cannot be cached. If the list is cached, it is directly displayed from the cache, although a translation from identifiers to label must still be performed. Each property in the list has a button to display its possible values; the values are not computed at the moment of loading the page to save time (a user is likely to only be interested in a fraction of the properties). This last process will be explained in Section 5.2.3.

5.2.2 Results entries

In the results section, the total number of documents of the results set and the metadata for the first fifty (ordered by ranking) are displayed. This section shows the main label of each result with its alt labels, description and image saved in the respective fields of the document for that resource; the image is scaled to a fixed height, because the images in the Wikidata triples are usually a high resolution image from Wikimedia. Clicking the label of one of the result entries redirects to the external Wikidata page of the resource.

It is important to note that if no alt labels, description or image were present for that resource, nothing will be displayed. However, if there is no label in the document, it defaults to the IRI without prefix: that is the `Q` or `P` value. This rule is applied for every translation from IRI to label, including the properties list of the navigation bar and facet selection.

The navigation bar and the result entries correspond to the Results Interaction in Section 3.1.3.

5.2.3 Facet selection

Next to every property in the navigation bar there is a button with the symbol +. Clicking the button sends a request to the back end to do a mixed query including the current query and the selected property to read every value for that property; if cached, these values are read from the cache for values and sent back to the front end directly. If the values are not cached, the `Q` values need to be translated before displaying them to the user.

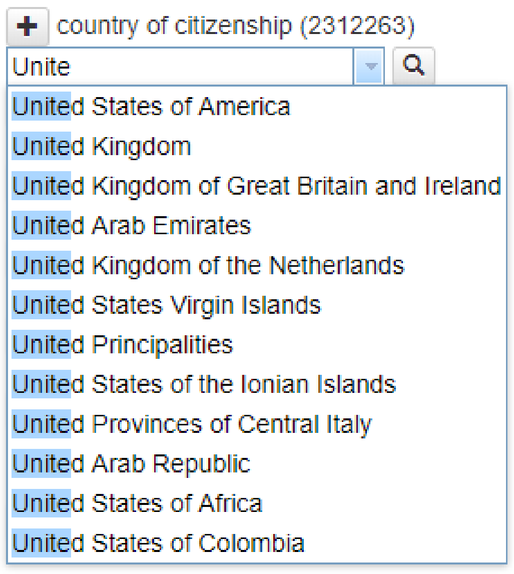

Figure 5.5: Value selection for a property

With the values returned, an autocomplete box is created. This autocomplete box provides a select-type input that shows all options (ordered by the respective ranking of the value) but the box also allows to write in the box and suggest the values that start with the input (as shown in Figure 5.5). Once a value is selected the query can be updated clicking the search button; this will send to the back end the current query plus the new facet(s) selected, resulting in loading again the results page but with a new query; each option in the autocomplete box shows its label but the value sent to the server is its `Q` value. This user interaction corresponds to the Facet Interaction described in Section 3.1.4.

Chapter 6

Evaluation

In this thesis, our main hypothesis is that the caching methods proposed in Section 4.5 enable interactive faceted browsing over large-scale, diverse RDF datasets. In this chapter, we present an evaluation over Wikidata to address the following related questions about our proposed faceted browsing system:

- How long does the index creation process take?
- How much space do the indexes use?
- Are the query runtimes acceptable?
- Does the caching help to improve the previous times?
- How are the usability and responsiveness of the system compared to current alternatives?
- Can the users perform successful queries on our system?

6.1 System Performance

All system performance measures were done using the Wikidata dump of 13 September 2017, which contains 1,771,601,730 triples describing a total of 74,114,172 different entities. The machine used has a processor Intel Xeon E5-2609 v3 and 32 GB of RAM.

6.1.1 Indexes statistics

As explained in Chapter 4, the system needs to store the information of the triples in two indexes and two caches. This version of the index stores the metadata of the entities in two languages: English and Spanish. Starting from the dump, the following processes are executed in order:

1. Initial index creation: described in Section 4.1.
2. Ranking computation: this process consists of performing the PageRank algorithm over the graph.
3. Data Index: the creation of the Data Index using the initial index and the ranking from the previous processes. (See Section 4.2)
4. The computation of the *need-caching list* described in Section 4.5.
5. Types index creation with the cache for properties: These two indexes are created at the same time (since certain query keys are types).
6. The creation of the cache for values.

Table 6.1: Times of all index-creation steps

Process	Time (s)	Time (hh:mm)
Initial index	22451	006:14
Ranking computation	15595	004:20
Data Index	2305	000:38
Need-caching List computation	15382	004:16
Types Index + Properties Cache	4356	001:13
Values Cache	386304	107:18

In order to create a new version of the index with a newer version of the dump, all processes need to be executed. The times for every step using the dump specified at the beginning are listed in Table 6.1. For this particular dump, the whole process took just over 5 days; however, newer dumps will have more triples and more entities, so this period can only be increasing over time. As well, adding more languages will impact the runtimes of the creation process, specifically for the initial index, the Data Index, and the cache for values.

Table 6.2: Size and documents stored by index

Index	Size (kB)	Documents
Data Index	5967412	74114172
Types Index + Properties Cache	5064	44413
Values Cache	1076404	16048

In Table 6.2 are the size and number of documents for every index; as before, newer dumps will increase in size and in number of documents for these indexes. The number of

documents in the Data Index corresponds to the total number of entities in the dump file. Due to the fact that the types index and the cache for properties are combined, the number of documents of the second index is the total of the types and query keys cached; for reference, the length of the need-caching list is 141 and the number of unique types is 44,272. The number of documents in the cache for values is the sum of the number of properties cached for every entry in the cache for properties; in spite of the size and the time building the indexes for caching took, it still refers to a small fraction of the whole dataset.

Note that is possible that a new dump may require changing the threshold that determined what properties needed a cache; this new threshold would have to be calculated and analyzed the same way as described in the corresponding section.

6.1.2 Browsing performance

To measure response times of the server when a user uses the system, several sequential requests are sent; the times, number of results and number of facets of each request were recorded. All browsing sessions in the test starts with a *type: human* query; then the session selects a random property from the first 20 in the properties list with a random value; the browsing session ends when the query returns only one result or there are no more properties to select. One thousand browsing sessions were executed with their respective requests to obtain the values of the properties, resulting in 2,943 requests to return results and 1,950 requests to return values for a property.

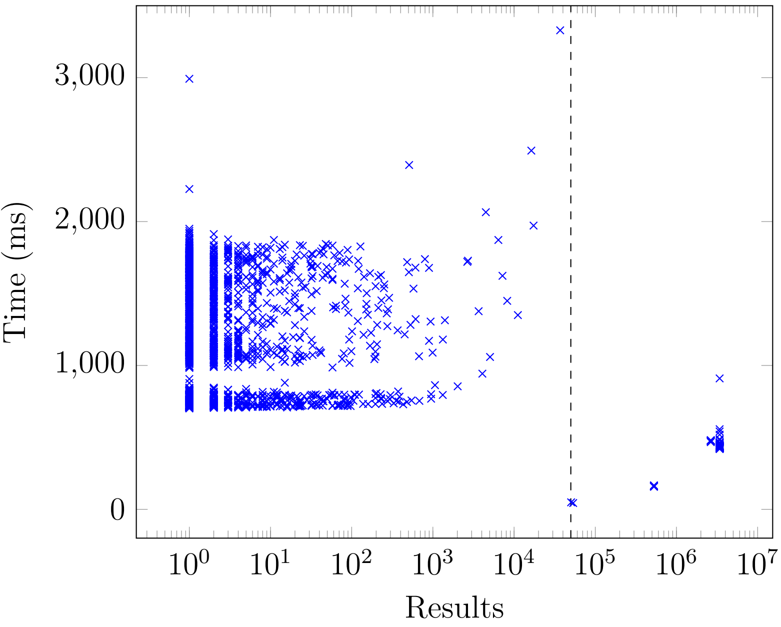

Figure 6.1: Times to load the Results page

Figure 6.1 depicts the times taken by the system to return a Results page based on the

number of results of the query. According to the graph, cached results (that is when the query has more than 50,000 results, shown to the right of the dashed line in the graph) have a response time of less than a second; otherwise, most non-cached results (left of the dashed line) took between one and two seconds to be returned to the user, reaching three seconds in the worst case. It is also worth mentioning that there are no queries between one million and two million results; there are two reasons for that: first, the process selects random properties and values, so it is probable that general facets (that return lots of results) were not selected; and second, the majority of facets will reduce the amount of results significantly. We look at this phenomenon in the following.

Table 6.3: Queries with one result based on the number of facets

Facets	Queries
0	0
1	404
2	387
3	116
4	48
5	22
6	20
7	2
No unique result	1

Table 6.3 shows how many facets the queries that return only one result have (not including type); there is one query that returns more than one result but it did not have more properties to select a facet; this is the result in the last row. In general, one or two facets will narrow down the results to one, but it is important to note that common facets (e.g. *gender: male*) will not and this random selection of facets will likely not choose one of them since such facets are relatively few. Still we see from Figure 6.1 that our experiment has queries that generate many results since we randomly select thousands.

Figure 6.2 shows the times specifically for the property requests (return all valid values for a property with respect to the current results) of the one thousand browsing sessions based on the number of available values for the given property. These times have a linear behaviour whether the values are cached or not; in any case, all times measured were below half a second, which we consider sufficient to support interactive browsing.

6.2 User study

For the next evaluation of the system, we gave 11 users individually the task to perform ten queries over the system this work presents and the query service provided by Wikidata (5 on

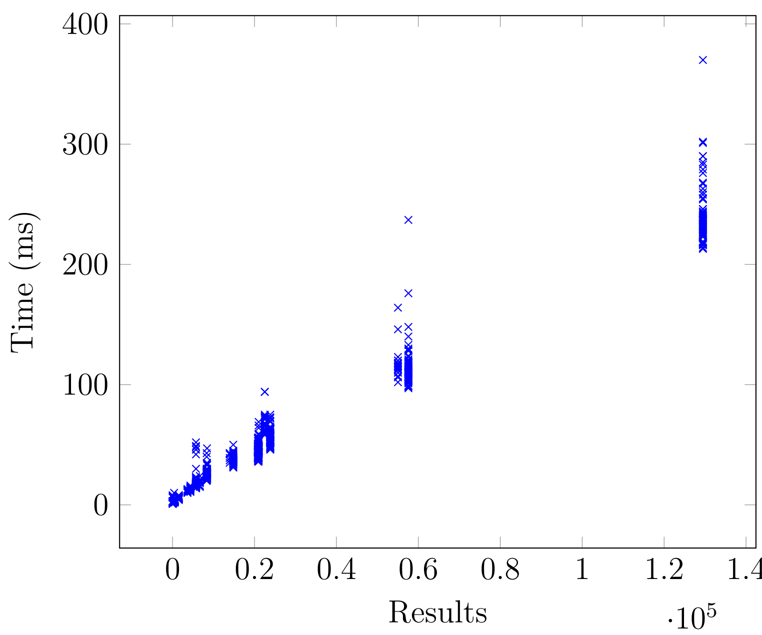

Figure 6.2: Times to load values for a property

each). Wikidata Query System[1] (see Figure 6.3) is a SPARQL endpoint with a user interface that provides suggestions of entities; the SPARQL query form is to the right and the filters of the query are to the left with a box that provides suggestions of entities (any entity can be entered, even if it does not make sense for the current query); the query in the figure is equivalent to the query *type: human* in our system.

The tasks we select for the comparison were adaptations of example queries provided by Wikidata (taken from the Example button in Figure 6.3); such tasks were ordered by increasing difficulty and the solution consists of a type and two facets at most, which the users need to infer based on the task description; no task would produce empty results and they could skip the task if they could not find the answer. The tasks (with their correct answers) were the following:

Question text	**Expected answer**
Plays	*(type:plays)*
Lakes in Cameroon	*(type:lake), (country:Cameroon)*
Lighthouses in Norway	*(type:lighthouse), (country:Norway)*
Popes	*(type:human), (position held:pope)*
Women born in Wales	*(type:human), (gender:female), (place of birth: Wales)*
Papers about Wikidata	*(type:scientific article), (main subject:Wikidata)*
Law & Order episodes	*(type:TV series episode), (series:Law & Order)*
Fictional characters from the Marvel Universe	*(type:fictional character), (from fictional universe:Marvel Universe)*
People dying by burning	*(type:human), (manner of death:death by burning)*
Mosquito species	*(type:taxon), (parent taxon:Culicidae)*

Half of the participants have to respond to odd questions on Wikidata and to even ques-

[1]https://query.wikidata.org/

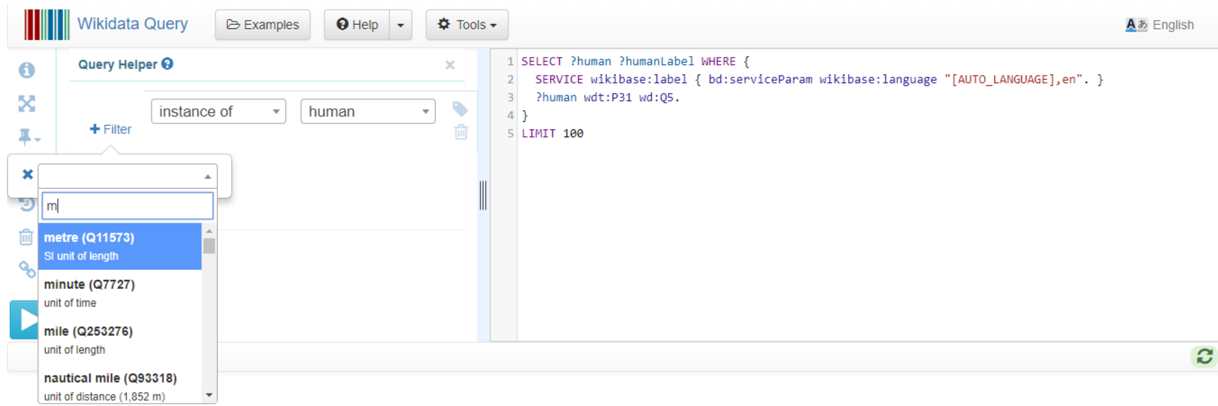

Figure 6.3: Wikidata Query Service

tions on our system; the other half respond to odd questions on our system and to even questions on Wikidata query service. All participants of the user study were students with knowledge of the Semantic Web and SPARQL, and with some familiarity with Wikidata; however, no details were provided on how to perform queries with the systems. Tasks were given in Spanish; the participant could use the systems in English or Spanish as they prefer, but the majority of the users prefer using the system in Spanish.

Participants have to register details of the solutions of every task to see if they were able to perform the query successfully. After the participants complete the tasks, they have to complete a questionnaire comparing the two systems; the questions were about how responsive and intuitive to use are the systems in a scale from 1 to 7.

Table 6.4: Tasks results of the user study

	System	
Answer	This system	Wikidata Query Service
Correct	23	37
Incorrect (uses keyword)	10	0
Incorrect (wrong type/facets)	2	10
Incorrect (other reasons)	1	1
Empty	19	7

The common errors of the user while performing the queries were as follows: In Wikidata, the users often did not select the correct predicate; that is probably because adding a filter to a query in the Wikidata query service automatically suggests a predicate based on the object, not selecting the correct one for the task. In our system, the participants had problems to select a type to start the browsing session; this user study was performed with a previous version of the system where boxes for type and keyword were displayed at the same time, causing some users to start the session with a keyword instead of a type, which made their

answer automatically incorrect; also, the previous version of the system allowed users to start a session even if no valid type was provided (because the keyword option was available too), resulting in users not waiting for the type suggestions to appear, leaving the solution for the task empty. Type selection is not an easy task when users do not know what can be a type; for example, pope is not a type (the correct type is person), but lighthouse is a type even when building is another type. The details of answers to the task including the reason of incorrect answers are given in Table 6.4.

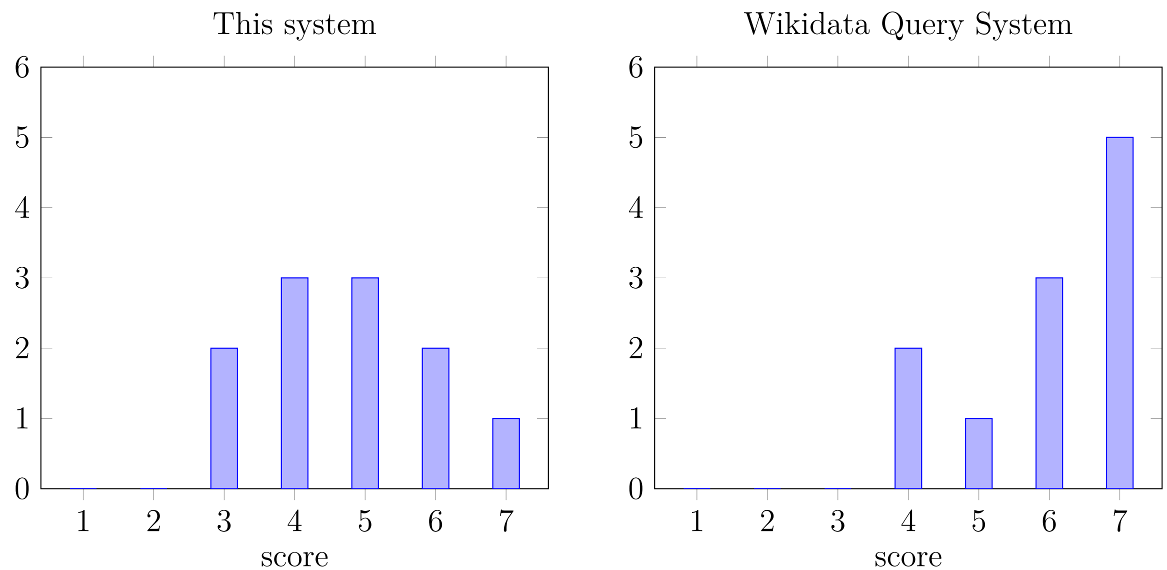

Figure 6.4: Responsiveness score according to users

The results were not very successful for the system presented in this work, which had fewer correct answers when compared to the Wikidata system. In Figures 6.4 and 6.5 are given the answers of the questionnaire; in the usability scale, our system scores on average 4.5 out of 7 (against the 5.5 of Wikidata's) and in the responsive scale scores 4.7 (in which Wikidata system scores 6). In the comments section, the main issue of this system, across all users, was the autocomplete options, where the users did not understand that a type must be selected in order for the system to return results, or the type they were thinking of was not available in the autocomplete options.

This first user study allowed us to improve the system by resolving some superficial but crucial issues we discovered; the two main improvements to the system were: separating keyword and type searches, and restricting the type search to only allow valid types displayed in the suggestions list; such improvements are included in the description of the system of the previous chapters. With this new version of the system a new study was performed which we now describe.

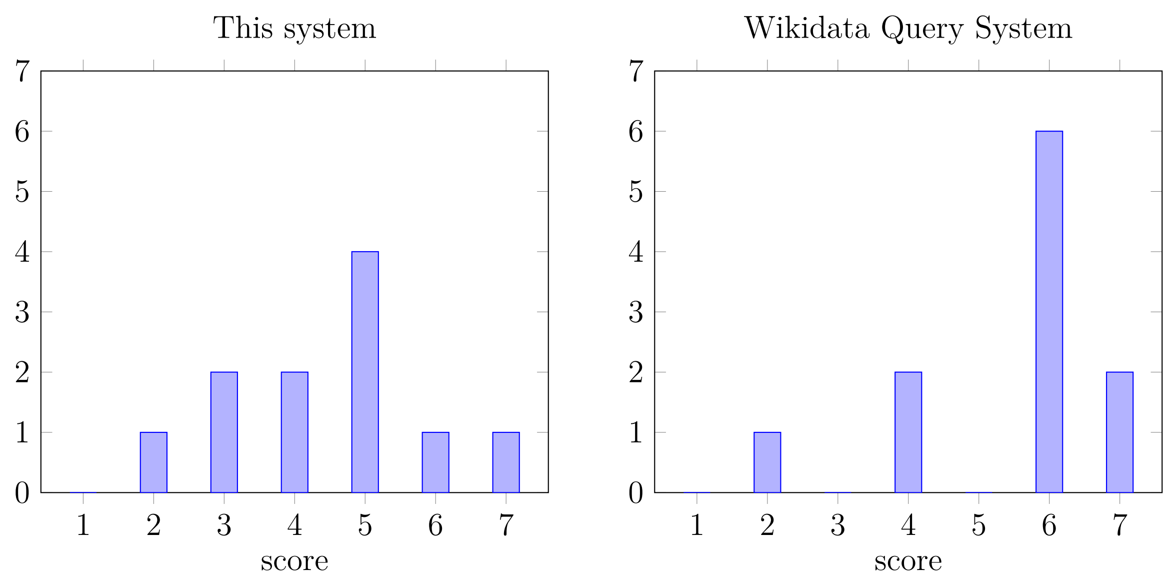

Figure 6.5: Usability score according to users

6.3 Wikidata's collaborators study

We sent a questionnaire with a link to our prototype and a list of 12 opinion-seeking questions to the Wikidata mailing list. Nine Wikidata collaborators volunteered to try the system and give feedback. The volunteers could freely use the system and respond to a new questionnaire afterwards. They have to select of a scale from 1 to 7 if they agree or not with the sentence, where 7 is strongly agree and 1 is strongly disagree. The results are presented in Figure 6.6. The reception was better than the previous user study; a question not shown in the figure was if they would use the system in the future: 4 of the 9 participants said yes, the other 5 said maybe; the participants expressed they like the system in the comments section and how the system could be improved. The most requested features for the system were the inclusion of datatype values, nested facets and subtypes (possible inclusion of these features are discussed in Section 7.1).

According to the results of Figure 6.6, the participants consider the system useful and it offers something novel for querying Wikidata; the ranking of the results was considered good by the evaluators but the ranking of the facets they found less intuitive. For the other answers the opinions were mixed; it is important to note that we do not know what queries they used to test the system, so that is a possible reason for different experiences and thus different evaluations about the loading times. About the question regarding if the system is easy to understand, some participants wrote as comments that it is not easy to start a session and they suggested to add some examples or placeholders in the boxes to better guide the users in how to use the system. Overall, the comments suggested that users believed the system was useful and offered a novel way to query Wikidata, though there is still room for

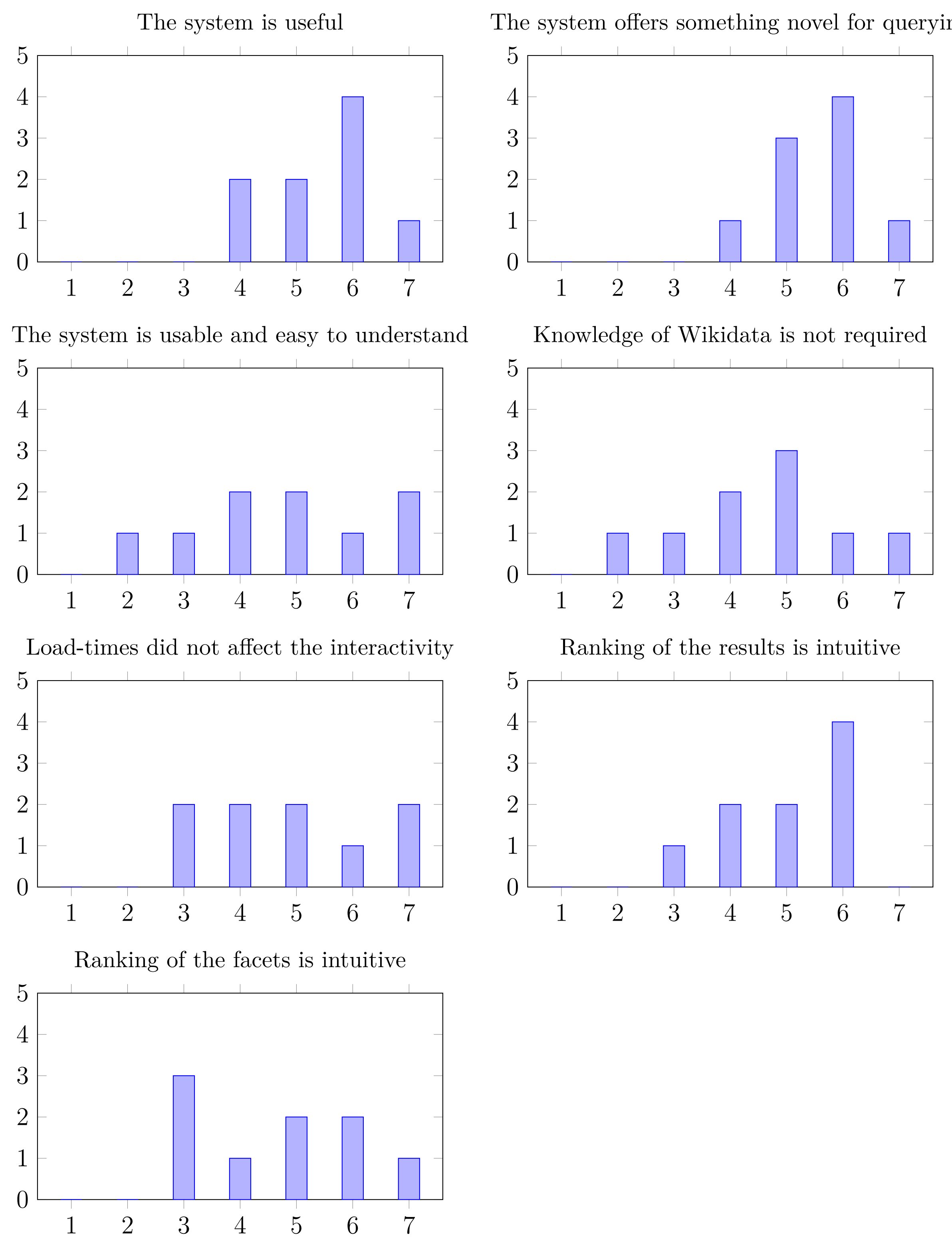

Figure 6.6: Answers of Wikidata's collaborators questionnaire

improvement in future work (discussed in Section 7.2).

6.4 Discussion

At the beginning of the chapter, we presented some questions that we were trying to answer with the results obtained from the three studies presented. The first questions were about the indexes: the total size used in indexes is about 7 GB, where 6 GB are used for the main Data Index, which stores 74 million documents; the remaining space is used in the caching, which is a reasonable size considering that the possible combinations of property-value pairs are quite a lot even for 141 cached queries. The total time to build the index and perform caching is above 5 days, where 4.5 days are used exclusively for the values cache; without caching the time required drops to 12 hours; despite that, the cache is needed to lower response times for large results sets, as we now discuss.

The next questions were about runtimes of the system: without caching (see Table 4.1) some queries would take minutes to complete, but adding the caching all times were below a couple of seconds. The caching reduces runtimes a lot, where cached queries reduce their time to below a second, so the time required to built the cache is worthwhile. Non-cached queries can take between one and two seconds, which is reasonable considering the large amount of documents in the cache and that some queries have to be excluded from the cache; the time-consuming operation for non-cached queries is the making and translation of the properties list, because the metadata extraction of the first fifty results is performed even for cached queries.

The final questions that this evaluation looked to answer to were about the user's experience with the system. Comparing the answers for the first user study with the second one, the usability of the system increased after fixing the superficial issues found in the first study, where Wikidata's collaborators score higher in usability than the test with students. Responsiveness of the system still has mixed opinions; the runtimes are probably not the explanation since we saw in Figure 6.1 that these are generally within a second; another possible explanation is the slow-paced process to obtain results (wait for the type suggestions, then wait for the values after a property is selected); however, this approach was taken to prevent suggesting empty-results to the user, so the question is: will the users prefer a faster-paced system where their suggestions might produce no results or the current guidance the system offers to select facets and obtain results. Nevertheless, the system still needs to better explain how to be used; both user studies have the comment that it is hard to understand the system at first and some examples could help to teach newcomers how to use this system.

Chapter 7

Conclusion

This work presented a new faceted browsing interface using the Wikidata dataset; this interface does not use a SPARQL endpoint, but instead uses indexes to store and search over the data to allow the computation of properties and values; this technique requires caching certain queries that would return more than 50,000 results in order to keep reasonable response times. The complete creation of the indexes and caches for Wikidata can take a week and uses about 7 GB of memory for two languages; times for queries will not take more than one second for cached queries and four seconds for non-cached queries (though most will execute faster), where the observed slowest queries at a few seconds should be contrasted with the slowest queries without caching, which require minutes. On the other hand, the size of the cache remains relatively small by selecting only the non-keyword queries with more than 50,000 results.

User response to the system was mixed: the consensus was that the system offers a new way to query RDF datasets with an intuitive selection of facets; however, the initial type search was not intuitive enough if the user has no prior knowledge of what is a type or what are the available types; the system requires a type selected from a list after they write a prefix and if there are no matches a browsing session cannot be started. Selecting a facet afterwards was simpler because the properties are already given to the user, as well the values for a selected property; unlike for the type selection, the users can see all properties available and their values.

7.1 Current limitations

The system considers as facets a pair of property and value where that value is a Wikidata resource; this is a current limitation that suggests some requested features for the system: datatype values, nested facets and subtype reasoning. First, properties whose values are datatype values (numbers, dates, coordinates) are not eligible as facets; the reason for this is that the system stores the values of a property in a string field of the document, where

its contents can be queried only by exact (or prefix) match character by character; in that case, the user could not add facets where the value is within a range (for example: museums between certain coordinates or humans with a date of birth in certain decade) because the facet would require the specific value, which is not viable for the user-interface. A naive solution would be to have a special field for every property with datatype values to store them in the documents, and thus, allow range queries; however, this approach would increase significantly the complexity of the system: first, the system would require a full analysis to determine which datatype values are important enough to have an exclusive field; second, the system would require a user interface with a range query selection between valid values to avoid empty results; third, it would not be possible to cache range queries with many results (e.g. people born between 1800 and 2000) since there would be too many possible range values.

The inclusion of nested facets is the next requested feature that is limited by the current structure of documents; nested facets allow to select as a value for a property a faceted search (for example: people whose father is a politician; the value for the property *father* is a new faceted search with the facet *occupation: politician*); the values for the properties have to be a specific resource, not a certain entity that has certain facets. This kind of query is very easy to perform in SPARQL, but not in our system. There is no easy way to add this feature because it would require to pre-compute all resources that match the nested facet and then add every value to the query with the logical operator OR; this approach is not reliable because if the nested facet contains millions of results, combining millions of queries might take too long, affecting the usability of the system.

The final feature limited by the structure of the system is subtype reasoning; this feature expects that when a type search is performed, the results include entities whose type is a subtype of the selected type. For example, the type of *apple* is *pome* and its supertype is *fruit*, *fruit* is a subtype of *food*; so *apple* should be included in the results of *type: fruit* (one level of subtype) and *type: food* (two levels of subtype). Although one level of subtype is easy to include in the system, more levels of subtype reasoning is a real challenge because determining all subtypes for a single entity is a recursive process; more problematically the possible results for general types (e.g. food) would return a massive amount of entities that, then again, the caching should consider.

7.2 Future work

Besides investigation on how to add the features described in the previous section, there are other possible future directions for this system and on the topic of querying RDF datasets. Note that all improvements to the system would require to keep the core of the system, which

is a user-friendly interface that does not require expertise on RDF and offers reasonable times for a user application. The first topic is to improve the main issues of the current system regarding the type selection; this could be fixed with an example query or providing a more intuitive interface trying to guide the user as to what are the available types and what could be a type.

Next, there is the topic of adding more flexibility to the facets, such as multivalue facets, where the user can select multiple values for a given property (e.g. people with citizenship in Chile and the USA), although certain restrictions to the number of values would be needed to not overload the query; existential facets, requesting that the items have some value for a given property (e.g. people with some award), so facets would not be restricted to a specific value. These kinds of queries are possible to implement with the current structure; however, these new ways to create queries would result in more queries with lots of results, so new caching techniques would need to be created to preserve the ideal execution times for a responsive system.

Finally, the amount of data in RDF is rapidly increasing, so the natural question is: will the structure of the system keep up with bigger datasets? More resources means more documents and thus more results to certain queries, causing an increase in both the cache's size and creation time; how can the system be improved to support even larger datasets (which will be normal in the up coming years); does the amount of properties to cache converge at some point, or will the cache grow so much that it would become unsustainable. These are some of the questions and challenges this system could face in the future.

Building usable interfaces for querying large-scale datasets that assume minimal user expertise or specific data structures, and that can offer responsive runtimes, remains a challenging problem. This thesis has established a first proof-of-concept for faceted browsing over RDF datasets with billions of triples, and has revealed some interesting research directions towards making such a system more usable, more efficient, more scalable and able to express more complex queries in future. We have named the final system GraFa (Graph Facets). The version loaded with Wikidata is publicly available at `http://grafa.dcc.uchile.cl`.

Bibliography

[1] M. Arenas, B. C. Grau, E. Kharlamov, Šarūnas Marciuška, and D. Zheleznyakov. Faceted search over RDF-based knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37–38:55 – 74, 2016.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[3] D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF Schema, February 2004.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[5] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, Mar. 2013. `http://www.w3.org/TR/sparql11-query/`.

[6] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space (1st Edition)*, volume 1 of *Synthesis Lectures on the Semantic Web: Theory and Technology*. Morgan & Claypool, 2011. Available from `http://linkeddatabook.com/editions/1.0/`.

[7] P. Heim, J. Ziegler, and S. Lohmann. gFacet: A browser for the web of data. In *Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW 2008)*, volume 417 of *CEUR-WS*, pages 49–58, 2008.

[8] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[9] D. L. Mcguinness and F. van Harmelen. OWL Web Ontology Language Overview, February 2004.

[10] O. Mishali, T. Yahav, O. Kalinsky, and B. Kimelfeld. elinda: Explorer for linked data. *CoRR*, abs/1707.07623, 2017.

[11] G. Schreiber and Y. Raimond. RDF 1.1 Primer. W3C Working Group Note, June 2014. `http://www.w3.org/TR/rdf11-primer/`.

[12] C. Stadler, M. Martin, and S. Auer. Exploring the Web of Spatial Data with Facete. In *Companion proceedings of 23rd International World Wide Web Conference (WWW)*, pages 175–178, 2014.

[13] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[14] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.