Bachelor Thesis

QLever UI: A context-sensitive user interface for QLever

Julian Bürklin

Albert-Ludwigs-University Freiburg

Department of Computer Science

Chair for Algorithms and Data Structures

September 7th, 2021

Author: Julian Bürklin

Examiner: Prof. Dr. Hannah Bast

Writing period: 09.06.2021 – 09.09.2021

Declaration

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

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

Freiburg den 07.09.2021

Place, Date

Signature

Abstract

QLeverUI is a web user interface for QLever [1], an efficient query engine developed at the chair for algorithms and data structures at Albert-Ludwigs-Univerity Freiburg. The interface mainly consists of an input area where one can enter SPARQL queries which are then sent to a QLever back end. After evaluating the query, the query results will be returned by the back end and displayed in tabular form. The main focus of QLever UI lies in supporting the user with writing these SPARQL queries. It provides autocompletion and next word suggestion functionalities, both in a context-sensitive manner. To do this, QLever UI queries the same QLever back end that is also used for the user's queries.
In the following we will explain the basic architecture and functions and the key components of our user interface.

Zusammenfassung

QLever UI ist eine Web-Benutzeroberfläche für QLever [1], eine effiziente Query-Engine, die am Lehrstuhl für Algorithmen und Datenstrukturen an der Albert-Ludwig-Universität Freiburg entwickelt wird. Die Benutzeroberfläche besteht hauptsächlich aus einem Eingabefeld, in das SPARQL-Abfragen eingegeben werden können, die dann an ein QLever Backend gesendet werden. Nachdem die Abfrage ausgewertet wurde, werden die Ergebnisse der Abfrage vom Backend zurück gegeben und in tabellarischer Form angezeigt. Das Hauptaugenmerk von QLever UI liegt darin, dem Benutzer beim Schreiben dieser SPARQL-Abfragen zu helfen. QLever UI bietet Funktionalität für Autovervollständigung und Vorschläge für das nächste Wort. Beide dieser Hilfen sind dabei kontextsensitiv. Um das zu realisieren, sendet das QLever UI Abfragen an das selbe QLever Backend, das auch für die Abfragen des Benutzers verwendet wird.

Nachfolgend werde ich die grundlegende Architektur, die Funktionen und die wichtigsten Bestandteile unserer Benutzeroberfläche erläutern.

Contents

1 Introduction **1**

1.1 Knowledge Bases 2

1.2 SPARQL and RDF 2

1.3 QLever 4

2 Related Work **5**

2.1 Wikidata Query Service 5

2.2 Broccoli 7

2.3 Yasgui + Gosparqled 8

3 Using QLever UI **10**

3.1 The interface 10

3.2 Next word suggestion and autocompletion 12

3.3 Entity name tool-tips 13

4 Configuring QLever UI **17**

4.1 The admin interface 17

4.1.1 Adding and configuring users 17

4.1.2 Adding and configuring example queries 18

4.1.3 Adding and configuring back ends 18

4.1.4 Back end defaults 24

5 Approach **27**

5.1 Evaluating the SPARQL template language 27

5.1.1 Warm up queries and warm up query patterns 27

5.1.2 Other placeholders 28

5.1.3 IF statements 31

5.2 Past approaches 35

5.2.1 The naive approach 35

5.2.2 Adding names to suggestions 36

5.2.3 Fully configurable queries 38

6 Discussion **39**

6.1 Conclusion 39

6.2 Future Work 40

6.2.1 Finding connected lines 40

6.2.2 Fully configurable name queries for name tool-tips 40

Bibliography **43**

List of Figures

1 RDF data visualized as a directed, labeled graph 3

2 Wikidata Query Service: Query Helper 6

3 Wikidata Query Service: OpenStreetMap view 7

4 A screenshot of the Broccoli UI 8

5 Yasgui 9

6 QLever UI: user interface 14

7 QLever UI: Query analysis 15

8 QLever UI: Suggestions 16

9 The admin interface 18

10 Configuring users 19

11 Configuring example queries 26

1 Introduction

There are many search engines that are easy to use and yet are powerful enough for most everyday searching. Consider for example Google search[1], one of the most popular search engines. The interface of Google search mainly consists of only two elements: An input field where one can enter a search term and a button to start searching and showing the results page. When clicking the search button, Google will, in very simplified terms, split the search term into keywords and show those results that contain at least one of the keywords. This simplicity, however, also comes with some disadvantages when searching for more specific things. Imagine we needed a list of actors who were born in Canada and can speak German. Google will find many[2] results for a search term like "Actors who were born in Canada and speak German". Most of those results however will not cover the exact topic we were searching for. The result list will contain entries which omit at least one of the restrictions of our query: There will be celebrities who are no actors, actors who were not born in Canada and actors who don't speak German. This is due to Google splitting our search term into keywords and not understanding the semantics of the query. But just because a conventional search engine does not show the information we are asking for, doesn't mean that this information does not exist. The Wikipedia page of an actor may very well list the spoken languages and the residence of this actor. You could therefore skim through as many Wikipedia pages of actors as possible

[1] https://www.google.com
[2] To be precise: 117.000.000 at the time of writing

and manually compile the list you are looking for. Of course, there must be a better way.

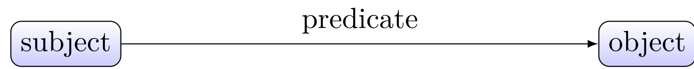
1.1 Knowledge Bases

A knowledge base is a collection of knowledge or facts about the world. These facts are stored in a way that may not be easily read or understood by humans. Knowledge bases are used by software and computer systems to store information in a structured way. The facts that such knowledge bases store can be restricted to certain topics (e.g. geo-location data of countries, cities and points of interest), but they don't need to be. As they store information about such a wide variety of topics, knowledge bases can grow quite big. One well-known knowledge base is Wikidata. Wikidata acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others [2]. Wikidata consists of information about more than 94 million items [3]. Searching through such knowledge bases is facilitated by different tools and query languages such as QLever and SPARQL.

1.2 SPARQL and RDF

SPARQL is a query language for data that is stored in the Resource Description Framework (RDF) format[4]. RDF is a framework for representing information in the web [5]. RDF data is basically structured as *<subject>* *<predicate>* *<object>* triples. Each of the entities inside such a triple is usually denoted as an IRI or a literal. This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes [6]. See Figure 1 for an example.

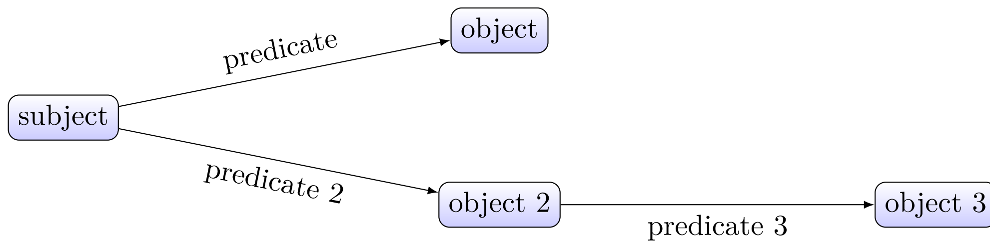
<subject> *<predicate>* *<object>*



(a) A single subject-predicate-object triple visualized as a directed, labeled Graph

---

<subject> *<predicate>* *<object>*
<subject> *<predicate 2>* *<object 2>*
<object 2> *<predicate 3>* *<object 3>*



(b) Three triples visualized as a directed, labeled Graph

Figure 1: RDF data visualized as a directed, labeled graph

SPARQL syntactically reminds of SQL. However, SPARQL queries make use of the same triple-notation as RDF. Consider the search term introduced in Chapter 1: "Actors who were born in Canada and speak German". A SPARQL query expressing this search could look like the following:

```
SELECT ?actor WHERE {
    ?actor <profession> <Actor> .
    ?actor <place_of_birth> <Canada> .
    ?actor <language_spoken> <German> .
}
```

This simple query consists of two sections: the so-called *select query* and *where clause* [7]. The *select query* describes which data should be incorporated in the result set.

The *where clause* describes the conditions that need to be met for an entry to end up in the result set. The SPARQL query displayed above assumes a simplified knowledge base where the contained entities are denoted by their name rather than an IRI.

1.3 QLever

There are search engines besides Google, Bing and the like which were designed to process queries with semantic information. One of those search engines is called QLever. QLever is a query engine for efficient combined search on a knowledge base and a text corpus [1]. The query language that is used is SPARQL. QLever is developed at the Chair for Algorithms and Data Structures at the Albert-Ludwigs-University Freiburg. QLever supports large data sets like for example the full Wikidata. In addition to standard SPARQL queries, QLever also provides extra functionality that enables queries to search in a text corpus like for example ClueWeb[3]. One downside of QLever, compared to the search engines mentioned above, is that it is less simple and less intuitive to use. The web interface that comes with QLever is just a plain text area where the user can type his query. Query results are then displayed in a table. QLever does not provide any form of assistance in writing the queries. This means that a user has to be well aware of the underlying ontology QLever is using. The user needs to know (or find out) how entities in the ontology are named and how to use them or he will not be able to formulate a correct query. This is why we created QLever UI. The goal of QLever UI is to make writing SPARQL queries for QLever simpler and more efficient by providing functionalities like autocompletion and next word suggestion.

[3]https://lemurproject.org/clueweb12/

2 Related Work

2.1 Wikidata Query Service

The Wikidata Query Service [8] is a web interface that lets a user write SPARQL queries. These queries can then be executed on the Wikidata knowledge base. The interface of the Wikidata Query Service comes with several tools that are designed to help users write queries without needing in-deep knowledge of the underlying knowledge base. Tools and features of the Wikidata Query Service include:

(a) **Query Helper** The Query Helper is a visual interface which uses buttons, drop downs and input fields to display and write SPARQL queries. When using the Query Helper, users do not need to know SPARQL in order to be able to write queries. Editing a query is done by clicking buttons and searching for the desired predicates and entities.

(b) **Autocompletions** Autocompletions can be invoked by pressing ctrl + space while typing in the query editor. Completions are available for SPARQL keywords as well as predicates and entities from the knowledge base. Auto completions for predicates and entities are not context-sensitive.

(c) **Examples** An extensive list of example queries that can be loaded into the SPARQL editor at any time. The example queries are categorized by topic and can be altered freely after loading them.

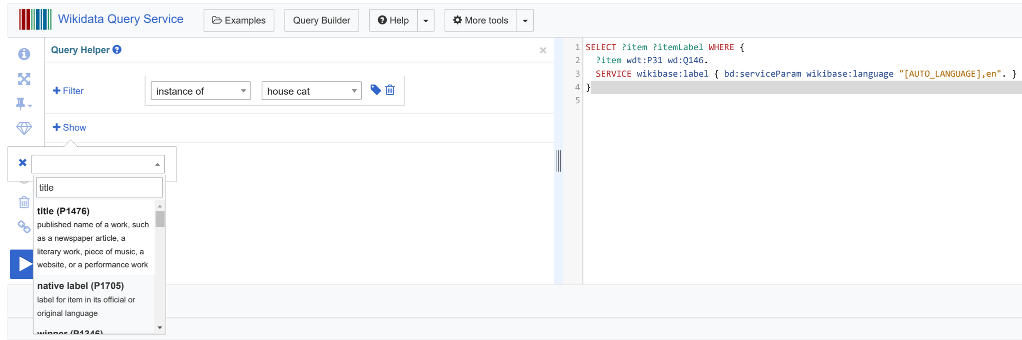


Figure 2: Editing a query using the Wikidata Query Service Query Helper

(d) **Tool-tips** Hovering over predicates and entities in the query editor will show a small tool-tip stating the name and a short description of the hovered-over object.

(e) **Format query** Can format the typed query with the click of a button in order to make it easier to read. This will also automatically fill in prefixes to entities when available.

(f) **OpenStreetMap integration** Query results that contain geo-location data can be rendered into OpenStreetMap, where each result row is depicted as a marker on the map.

(g) **Share** Creates a short link for the query you just typed, making it easy to share it with others.

(h) **Download** Query results can be downloaded as JSON, TSV, CSV or HTML file.

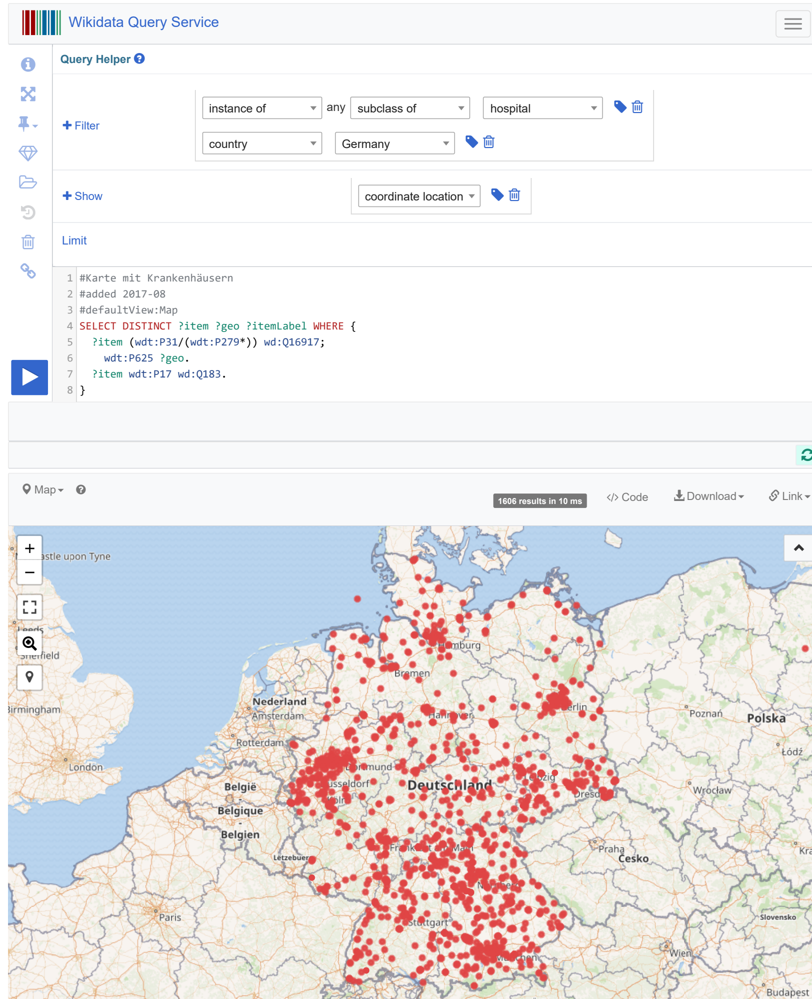


Figure 3: Visualizing a query that searches for hospitals in Germany in Open-StreetMap

2.2 Broccoli

Broccoli is a search engine that is able to perform full-text search combined with ontology search, similar to QLever [9]. It features a quite different user interface than QLever UI or the Wikidata Query Service, however. The interface is designed to be intuitive and easy to use even if a user does not know about query languages like SPARQL. Broccoli only supports a subset of SPARQL, namely trees with a single free variable at the root. Broccoli has only a small input field where the user

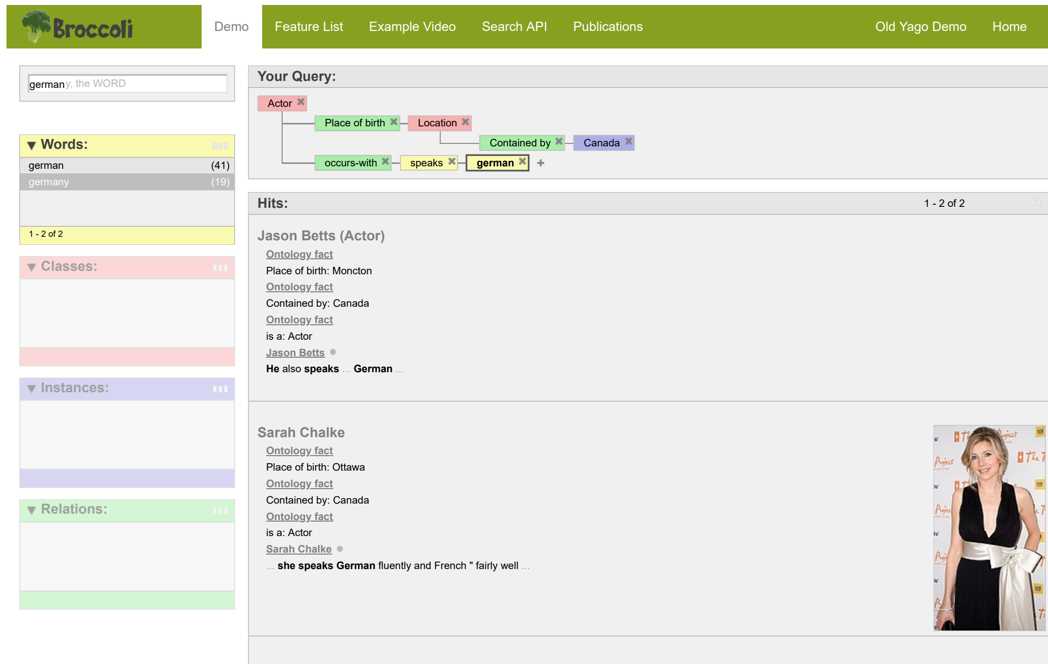


Figure 4: A query in Broccoli: Actors who were born in Canada and speak German

can input text. The four boxes below this input field will display suggestions for words, classes, instances and relations. The suggestions are aware of the current query context and, if something was typed into the search input, will complete the typed word. The current query is visualized as a tree where each node can be deleted and / or changed by clicking on it and using the search input and the four proposal boxes on the left. Query results are not displayed in tabular form like in QLever UI and Wikidata Query Service. Results are instead displayed as list, where each element of the list is titled with the found entities name and an image, if there is one. The list elements also contain ontology facts and text excerpts as evidence.

2.3 Yasgui + Gosparqled

Yasgui [10] is a SPARQL user interface that looks similar to QLever UI. Like QLever UI, its query editor is built using CodeMirror. Yasgui provides suggestions for prefixes,

Query

https://dbpedia.org/sparql

```
SELECT ?label ?loclabel WHERE {
  ?museum a <http://dbpedia.org/ontology/Museum> .
  ?museum <http://www.w3.org/2000/01/rdf-schema#label> ?label .
  ?museum <http://dbpedia.org/ontology/location> ?loc .
  ?loc <http://www.w3.org/2000/01/rdf-schema#label> ?loclabel .
  FILTER(lang(?label) = "en") .
  FILTER(lang(?loclabel) = "en")
}
LIMIT 10
```

10 results in 0.11 seconds

Compact | Filter query results | Page size: 50

	label	loclabel
1	"Baltimore Museum of Art"@en	"Baltimore"@en
2	"Baltimore Museum of Industry"@en	"Baltimore"@en
3	"Baltimore Streetcar Museum"@en	"Baltimore"@en
4	"Burhaniye Natio ... Culture Museum"@en	"Balıkesir Province"@en
5	"Bangkok Art and Culture Centre"@en	"Bangkok"@en
6	"Bangkok Folk Museum"@en	"Bangkok"@en
7	"Bangkok National Museum"@en	"Bangkok"@en
8	"Bangladesh National Museum"@en	"Bangladesh"@en
9	"Bannu Museum"@en	"Bannu District"@en
10	"Baoji Bronzeware Museum"@en	"Baoji"@en

Showing 1 to 10 of 10 entries < 1 >

Figure 5: A query in Yasgui: Museums and their locations

used variables and entity autocompletions. By default it has no support for next word suggestions. The default autocompletions are provided via the Linked Open Vocabularies API [11]. These suggestions are not context-sensitive. However, Yasgui can be customized to use other autocompletion logic. This is where Gosparqled [12] comes into play. Gosparqled provides the functionality for context-sensitive autocompletions and next word suggestions. It can be easily integrated into Yasgui. It does not support suggestions that consider canonical or alternative names of entities. Results are displayed in tabular form.

3 Using QLever UI

In this chapter we will describe the user interface of QLever UI. At first we will show and explain the different parts of the user interface (Section 3.1). Then we will explain the features of QLever UI, including context-sensitive suggestions and autocompletion (Section 3.2).

3.1 The interface

The user interface consists of the query editor and a number of input fields and buttons. These elements are described in this section. For reference, see Figure 6.

(a) **Query editor** The query editor lies in the center of the user interface. It is a big text area in which SPARQL queries can be typed. The Query editor supports SPARQL syntax highlighting.

(b) **Query results** Query results are displayed below the query editor. Each result is displayed as a single row in a table. Above the result table is shown additional information concerning the query result. This information consists of the time it took the QLever engine to compute the result and the total number of lines found.

(c) **Button bar** Below the query editor are several buttons:

1. Execute. Sends the query that is displayed in the query editor to the QLever back end. Query results will then be displayed in the "Query results" section below.

2. Download. Also sends the query to the QLever back end, but will download the result either as TSV or CSV file.

3. Share. Encodes the current query into both a short link and as query string. Both links are shown in a modal and can be copied to the clipboard for fast and easy sharing.

4. Reset. Clears the Query editor and reloads the currently selected back end.

5. Clear cache. Sends the command to clear the query cache to the current QLever back end.

6. Analysis. Displays the last executed query as a tree. Each node in the tree describes an operation the QLever back end needed to execute in order to get the query result. See Figure 7

7. Examples. Shows a list of all available example queries for the back end. Clicking a list item will populate the query editor with the corresponding query.

8. Mode select. Changes the suggestion mode in the query editor. Available modes are (1) keywords and variables only, (2) context-insensitive entities, (3) context-sensitive entities and (4) context-sensitive and insensitive mixed mode.

9. Clear cache before every request. Clears the cache each time before sending the current query to the QLever back end.

(d) **Back end select** Shows a list of all QLever back ends that were configured in QLever UI. Clicking an element will load the corresponding back end.

(e) **Index information button** Shows additional information concerning the index that is loaded in the QLever back end.

(f) **Back end information button** Shows settings for the currently selected back end.

(g) **Shortcuts / Help** Shows available keyboard shortcuts for working with the interface.

3.2 Next word suggestion and autocompletion

When writing a query using QLever UI, the UI will try to suggest what the next word in the query could be (next word suggestion) or how to complete the word the user is currently typing (autocompletion). These suggestions can take the form of keywords or constructs that are used by SPARQL, variables that are currently used in the query, or entities from the knowledge base the user is querying. The suggestions are displayed in a small box below the cursor. If the label of a suggested entity is known, it will be shown alongside the entity. See Figure 8.
QLever UI supports four different suggestion modes:

1. **Keywords and variables only** will only suggest SPARQL keywords and variables that are used in the query. No entities from the knowledge base will be suggested.

2. **Context-insensitive suggestions** will additionally suggest entities from the knowledge base. The user's query will not be taken into context for retrieving these suggestions.

3. **Context-sensitive suggestions** will suggest entities by taking the user's current query as context.

4. **Mixed mode** will simultaneously send a context-sensitive and a context-insensitive query to the QLever back end in order to retrieve entity suggestions. The context-sensitive query is sent with a predefined timeout. If this query returns a response before the timeout is reached, it's suggestions will be used. If the query times out, the results of the context-insensitive query are used.

3.3 Entity name tool-tips

Entities in RDF knowledge bases are usually denoted as IRIs. This can make reading and understanding queries quite cumbersome. In QLever UI, hovering the mouse over an entity IRI will automatically show a tool-tip displaying the canonical name of the entity. Names of previously suggested entities are stored in a list and retrieved as soon as the user hovers over an entity. If QLever UI does not know the name of an entity, it will be queried from the back end. This means that the name tool-tips will also work for queries that are copy/pasted into QLever UI.

QLever UI

Wikidata (d)

(e) Index Information (f) Backend Information (g) Shortcuts / Help

(a)

```
PREFIX schema: <http://schema.org/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?actor_name ?place_of_birth_name WHERE {
  ?actor wdt:P106 wd:Q33999 .
  ?actor wdt:P19 ?place_of_birth .
  ?place_of_birth wdt:P17 wd:Q16 .
  ?actor wdt:P1412 wd:Q188 .
  ?actor schema:name ?actor_name .
  ?place_of_birth schema:name ?place_of_birth_name .
  FILTER (LANG(?actor_name) = "en") .
  FILTER (LANG(?place_of_birth_name) = "en") .
}
```

(c) Execute | Download | Share | Reset | Clear cache | Analysis | Examples

4. Mixed mode

☐ Automatically add names to result

☐ Clear the cache before every request

(b) Query results:

7 lines found | 651ms in total | 651ms for computation | 0.039ms for resolving and sending

	?actor_name	?place_of_birth_name
1	Finlay Wojtak-Hissong	Canada
2	Bill Mockridge	Toronto
3	Benjamin Sadler	Toronto
4	Sarah Chalke	Ottawa
5	Nic Romm	Montreal
6	Anke Engelke	Montreal
7	Bruce LaBruce	Southampton, Ontario

Figure 6: The user interface of QLever UI

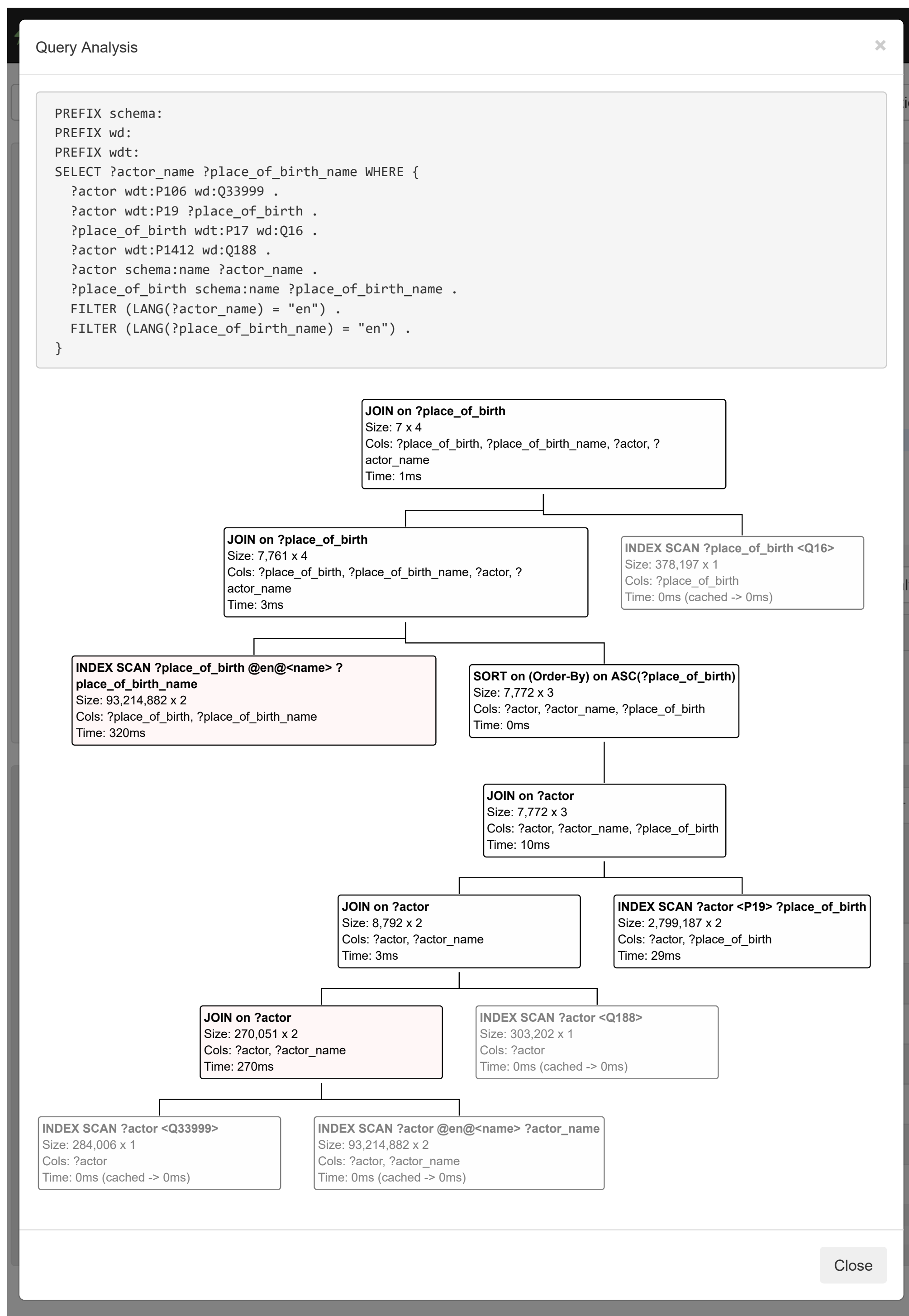


Figure 7: Query analysis in QLever UI

```
1 S
  SELECT  WHERE {

  }
```

(a) Suggesting a keyword

```
1 SELECT WHERE {
2    ?actor
3  }
        ql:contains-word
        ql:contains-entity
        wdt:P31                         "instance of"@en
        wdt:P1476                             "title"@en
        wdt:P577                     "publication date"@en
        wdt:P1433                        "published in"@en
        wdt:P2093                  "author name string"@en
        wdt:P304                            "page(s)"@en
```

(b) Next word suggestion

```
1 SELECT WHERE {
2    ?actor job
3  }
        wdt:P106                 "occupation"@en / "job"@en
        wdt:P2868       "subject has role"@en / "job title"@en
        wdt:P1366      "replaced by"@en / "next job holder"@en
        wdt:P1365   "replaces"@en / "previous job holder"@en
        wdt:P1043                            "IDEO Job ID"@en
        wdt:P7179 "Service d'Information sur les Etudes et l
        wdt:P1052           "Portuguese Job Code CPP-2010"@en
```

(c) Autocompletion with synonyms

```
1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 SELECT WHERE {
4    ?actor wdt:P106 wd:Q1650915 .
5   ?actor
6  }
        ql:contains-word
        ql:contains-entity
        wdt:P31                         "instance of"@en
        wdt:P1476                             "title"@en
        wdt:P577                     "publication date"@en
        wdt:P1433                        "published in"@en
        wdt:P2093                  "author name string"@en
```

(d) Mixed mode: context-insensitive suggestions

Figure 8: Different suggestions in QLever UI

4 Configuring QLever UI

QLever UI initially does not have any information about neither the QLever back ends it is supposed to be querying, nor their underlying knowledge bases. Many features like the suggestions for entities, names and prefixes need to be correctly configured before they start working. Only the suggestions for variables and SPARQL keywords are programmed to work without prior configuration.

4.1 The admin interface

QLever UI comes with an extensive admin interface which is not accessible for normal users. This interface serves to add and configure users, QLever back ends, and example queries.

4.1.1 Adding and configuring users

Users have the sole purpose of being able to configure QLever UI. If a user is created or configured, the only relevant fields are *Username*, *Password* and the *Active*, *Staff status* and *Superuser status* permission flags. If these fields are set, the user will be able to log into the admin interface and configure QLever UI.

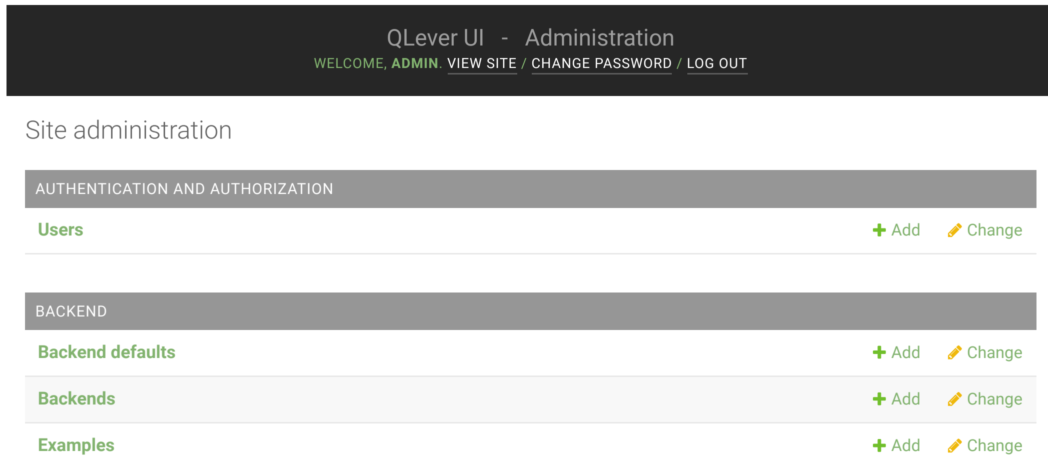


Figure 9: The admin interface of QLever UI

4.1.2 Adding and configuring example queries

In order for the QLever UI to show example queries, they need to be defined first. The example configuration consists of only three fields: The name, the actual query and the back end the example query belongs to.

4.1.3 Adding and configuring back ends

The most important thing to configure are back ends. Without first configuring a back end, QLever UI will not be able to write any SPARQL queries. This section will explain the most relevant settings that are available for back ends.

Autocomplete Queries

There are two sections for autocompletion queries. They are called *Autocomplete Queries (context-sensitive)* and *Autocomplete Queries (context-insensitive)*. Both of these sections contain three fields: *Subject*, *Predicate* and *Object autocompletion*

Figure 10: The relevant fields when configuring users

query. These settings are responsible for the entity suggestions to work properly. In each of these fields, a SPARQL query needs to be submitted that is capable of finding entities (subjects, predicates and objects, respectively) that should be suggested. As their names suggest, the queries in the *context-sensitive* category are used in the context-sensitive suggestion mode and as the main query in mixed mode, while the queries defined in the *context-insensitive* category are used in the context-insensitive mode and as backup query in mixed mode.

These queries are not written in standard SPARQL syntax, however. In order to be able to factor in the current user's query and the currently typed word, we devised

a simple template language that adds some statements and placeholder keywords to the syntax. These statements and placeholders will be parsed and evaluated by QLever UI when suggestions are to be queried. In this process, all the extra syntax will be removed or replaced, resulting in an ordinary SPARQL query that can be sent to the QLever back end.

The following statements and keywords can be used in these queries:

1. `%CURRENT_SUBJECT%` and `%CURRENT_PREDICATE%`

   Are replaced by the subject or predicate of the line the user is currently typing.

2. `%CURRENT_WORD%` and `%<CURRENT_WORD%`

   Are replaced by the word the user is currently typing. `%<CURRENT_WORD%` prepends a `<` if not already available.

3. `# IF #`, `# ELSE #` and `# ENDIF #`

   Text inside an `# IF #` or `# ELSE #` block will be removed if the given condition is not satisfied.

   Possible conditions are

`CURRENT_WORD_EMPTY`	true if the cursor position is at a space
`CURRENT_SUBJECT_VARIABLE`	true if `%CURRENT_SUBJECT%` is a variable
`CURRENT_PREDICATE_VARIABLE`	true if `%CURRENT_PREDICATE%` is a variable
`CONNECTED_TRIPLES_EMPTY`	true if `%CONNECTED_TRIPLES%` is empty

   Conditions can be connected through `AND`, `OR` and `!`

4. `%PREFIXES%`

   Inserts the prefix declarations the user has made in addition to all the prefixes that are defined in the back end settings.

5. `%CONNECTED_TRIPLES%`

   Inserts the lines of the user's query that are connected to `%CURRENT_WORD%`

6. `%PREFIXES%`

   Inserts every prefix used in the user's query, in addition to every prefix QLever UI was configured to suggest for this back end.

7. Placeholders for the queries defined below

   All the queries defined in "Warm up Queries" and "Warm up Query Patterns" can be inserted into an autocompletion query. The placeholder for a setting is the name of the setting in upper case, spaces replaced by underscores, all enclosed in percent signs.

Variable Names

This section defines various variable names that are used throughout the queries that were defined in the previous sections. In order to function properly, QLever UI needs to know which variables are used and what their meanings are. The following four variable types can be set:

1. **Variable for suggested entity** stores the actual entity IRIs that should be suggested by QLever UI.

2. **Variable for suggestion name** stores the canonical name of the suggested entities.

3. **Variable for alternative suggestion name** stores an alternative name of the entity. This name is only shown in autocompletions where the canonical name does not match the currently typed word, but the alternative name does.

4. **Variable for reversed suggestion** is set to 1 if a predicate suggestion is reversed.

For each of these settings, an arbitrary variable name can be chosen. These variable names then need to be used in the queries defined in the previous section, according to their description.

Autocomplete settings

The *Autocomplete settings* consist of four fields that change the general behavior of the suggestion engine.

The **Default suggestion mode** defines which of the four suggestion modes should be active when opening this back end in QLever UI. The users can still change the mode if they wish to. This is just the default value that is preselected.

Autocomplete timeout determines a timeout in seconds after which an autocompletion query should be aborted. If the timeout is reached, there will be no entity suggestions and an error message will appear. If set to 0, no timeout will be enforced. Although QLever supports setting query timeouts with a request, QLever UI enforces it's timeouts by itself. This is necessary because QLever's timeouts are not exactly precise. This could result in requests that can sometimes take a few seconds longer to abort than what was configured.

Mixed mode timeout is only used in the mixed suggestion mode. It is usually set to be shorter than de default timeout. In this mode, the main request with the context-sensitive query is sent with the *Mixed mode timeout*, while the backup request with the context-insensitive query is sent with the normal *Autocomplete timeout*.

Replace predicates in autocompletion context With this setting, some of the predicates used in the autocompletion queries can be replaced by other predicates. This feature is mainly used for knowledge bases which contain multiple labels in different languages for the same entities. With this, it is possible to replace language-agnostic predicates by language-aware ones. This leads to entities always being suggested with their name in only the desired language, instead of the same entity being suggested in multiple different languages

Warm up queries

QLever comes with an ability called cache pinning. This means executing a query and keeping it's result in the query cache so that future executions of this query can use the cached results instead of executing the query again. QLever UI leverages this mechanism by providing the possibility of defining up to five arbitrary queries per QLever back end that can be pinned to the query cache. These queries are called warm up queries. The idea behind this so-called "cache warming" is to take some of the most time-consuming and/or most frequent (sub-)queries that are needed for QLever UI's autocompletion queries and adding them to QLever's query cache even before they are executed in a real autocompletion context. This ensures that entities can be suggested as fast as possible. The warm up queries are written in the same SPARQL template language as the *Autocomplete Queries* above. QLever UI will parse and evaluate them before sending them to the QLever back end. Additionally, each warm up query can be addressed in any of the *Autocomplete Queries* with the placeholder `%warm up_QUERY_X%` where $X \in (1, ..., 5)$.

Currently, QLever UI supports a maximum of five warm up queries per back end. Adding more warm up queries can be done in a few easy steps and is further discussed in the `README.md` file in the QLever UI GitHub repository.

Warm up query patterns

Warm up query patterns are patterns commonly used in warm up or autocompletion queries. They are written in our SPARQL template language. These patterns do not need to be complete SPARQL queries. They can also consist of only a few triples or a sub query. These patterns can be substituted intowarm up or autocompletion queries using placeholders.

Frequent predicates

The *Frequent predicates* are used for another form of cache warming. In these fields, an arbitrary amount of predicates can be defined. The defined predicates are then each pinned to the cache with a simple query that looks like this

Listing 4.1: Pinning frequent predicates

```
1 SELECT ?x ?y WHERE { ?x <predicate> ?y }
```

The predicates can be pinned in unordered fashion like above or optionally can be ordered. Ordering them will result in two queries that look like this

Listing 4.2: Pinning frequent predicates with order

```
1 SELECT ?x ?y WHERE { ?x <predicate> ?y } ORDER BY ?x
2 SELECT ?x ?y WHERE { ?x <predicate> ?y } ORDER BY ?y
```

showing names

The *Showing names* section contains three fields called **Subject name clause**, **Predicate name clause** and **Object name clause**. These are used to retrieve the name of an entity when hovering over it in the query editor. If a name is found, it will be shown in a little tool-tip above the entity. These fields do not contain complete SPARQL queries but only the triples that are needed to retrieve the name of a given subject, predicate or object, respectively.

4.1.4 Back end defaults

Many of the settings can be configured similar or even the same between several different back ends. In order to make configuring a new back end easier and faster, we implemented a settings page for back end default settings in the admin interface.

The default settings contain all of the settings described in Section 4.1.3 and some more. If a default setting is configured, it's value will be displayed as gray placeholder text in the back end configuration (See Figure **??**). As soon as something is typed, the placeholder text will disappear and whatever was entered can be saved as the new setting. If the default configuration needs to be adapted for a particular back end, there is an *edit* button below each setting that uses the default configuration. By clicking this button, the default configuration will be loaded as normal text into the input field and can be edited.

QLever UI - Administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Backend › Examples › My example query

Change example

My example query

HISTORY

Backend:

Wikidata Full @ Uni Freiburg

Name:

My example query

Name of this example to show in the user interface

Query:

```
SELECT ?name WHERE {
  # example query goes here
}
```

QLever UI

Wikidata Full @ Uni Freiburg | Index Information | Backend Information | Shortcuts / Help

```
1 SELECT ?name WHERE {
2   # example query goes here
3 }
```

Execute | Download | Share | Reset | Clear cache | Analysis | Examples

4. Mixed mode

Automatically add names to result

Clear the cache before every request

My example query

Add current query as example

Add new example

Figure 11: Configuring and showing example queries

5 Approach

This chapter will cover our approach to the SPARQL template language we use for configuring the autocompletion queries. We will show how each of the keywords and statements is evaluated and substituted in an autocompletion query.

5.1 Evaluating the SPARQL template language

As described in Section 4.1.3 our template language adds the following elements to the SPARQL syntax:

1. Placeholders for warm up queries and warm up query patterns

2. Placeholders for `%CURRENT_WORD%`, `%<CURRENT_WORD%`, `%PREFIXES%`, `%CONNECTED_TRIPLES%`, `%CURRENT_SUBJECT%` and `%CURRENT_PREDICATE%`

3. `# IF #`, `# ELSE #` and `# ENDIF #` statements with their logical operators `AND`, `OR` and `!`

5.1.1 Warm up queries and warm up query patterns

In the first step, QLever UI substitutes the placeholders for the warm up queries and warm up query patterns by their values. This is done by iterating over all placeholder names that should be replaced and replacing them by their value using

regular expressions. This may be done several times, because every substituted placeholder may introduce new placeholders that need to be substituted.

```
1  function substituteCustomPlaceholders(query) {
2      let replacedQuery = query;
3      // WARMUP_AC_PLACEHOLDERS is a dictionary:
4      // placeholder names -> substitution value
5      for (const placeholderName in WARMUP_AC_PLACEHOLDERS) {
6          replacedQuery = replacedQuery.replace(
7              new RegExp(`%${placeholderName}%`, "g"),
8              WARMUP_AC_PLACEHOLDERS[placeholderName]
9          );
10     }
11     if (replacedQuery === query) {
12         return replacedQuery;
13     }
14     return substituteCustomPlaceholders(replacedQuery);
15 }
```

Listing 5.1: Substituting warm up query placeholders

5.1.2 Other placeholders

In the second step, all the other placeholders are replaced, leaving only the `# IF #` statements in the SPARQL query when finished. First, the `%CURRENT_WORD%` and `%<CURRENT_WORD%` placeholders are replaced. The current word is usually used for `REGEX` SPARQL filters. Some characters need to be escaped in order to not cause issues with the regular expression syntax in the resulting SPARQL query. After that, the substitution is again implemented through simple regular expressions.

```
function substituteCurrentWord(query, word) {
    word = word.replace('.','\\\\.').replace('*','\\\\*')
    .replace('^','\\\\^').replace('?','\\\\?')
    .replace('[','\\\\[').replace(']','\\\\]');
    const word_with_bracket = (
        (word.startsWith("<") || word.startsWith('"')) ? "" : "<"
    ) + word.replace(/'/g, "\\'");

    query = query.replace(
        /%<CURRENT_WORD%/g, word_with_bracket
    ).replace(/%CURRENT_WORD%/g, word);
    return query;
}
```

Listing 5.2: Substituting the CURRENT_WORD placeholders

Next, the `%PREFIXES%` keyword is substituted. The prefixes QLever UI is configured to suggest are appended to the prefixes that are used in the current query. This makes it possible to use these prefixes in autocompletion queries even when the user does not use them in the actual query.

```
function substitutePrefixes(query, prefixes) {
    for (const pName in COLLECTEDPREFIXES) {
        prefixes += `\nPREFIX ${pName}: <${COLLECTEDPREFIXES[pName]}>`
    }
    query = query.replace(/%PREFIXES%/g, prefixes)

    return query;
}
```

Listing 5.3: Substituting the PREFIXES placeholder

After that, the `%CONNECTED_TRIPLES%` placeholder needs to be substituted. The connected triples are passed to the following function as an array of lines. This

placeholder is captured together with the amount of white space that stands before it. The same amount of white space is also added to every line of the connected lines. This ensures that the resulting SPARQL query remains well readable, with the line indentations as defined in the autocompletion query.

```
function substituteConnectedTriples(query, connectedLines) {
    let match = query.match(/(\s*)%CONNECTED_TRIPLES%/);
    while (match != null) {
        query = query.replace(
            /%CONNECTED_TRIPLES%/g,
            connectedLines.join(match[1])
        );
        match = query.match(/(\s*)%CONNECTED_TRIPLES%/);
    }
    return query;
}
```

Listing 5.4: Substituting the CONNECTED_TRIPLES placeholder

The last keywords that need to be replaced are `%CURRENT_SUBJECT%` and `%CURRENT_PREDICATE%`. If the current predicate is in the form `<pred1>/<pred2>*`, it is replaced by `<pred1>|<pred2>` before being substituted.

```
function substituteCurrentVariables(query, words) {
    if (words.length > 0) {
        query = query.replace(/%CURRENT_SUBJECT%/g, words[0]);
    }
    if (words.length > 1) {
        pred = words[1].replace(/^([^ \/]+)\/([^ \/]+)\*$/, "$1|$2");
        query = query.replace(/%CURRENT_PREDICATE%/g, pred);
    }
    return query;
}
```

Listing 5.5: Substituting CURRENT_SUBJECT and CURRNE_PREDICATE

5.1.3 IF statements

In this last step, the `# IF #`, `# ELSE #` and `# ENDIF #` statements with their conditions and logical operators `AND`, `OR` and `!` will be evaluated. First, all the `# IF [condition] #` statements in the query are searched. Every match will be added to a list, with the additional information of the character index, length and condition of the statement.

```
function evaluateIfStatements(query, word, connectedLines, words) {
    // find all IF statements
    let if_statements = [];
    const ifRegex = /#\sIF\s+([!A-Z_\s]+)\s+#/;
    let match = query.match(ifRegex);
    let substrIdx = 0;
    while (match != null) {
        const index = match.index;
        const len = match[0].length;
        substrIdx += index + len;
        if_statements.push({
            'IF': { 'index': substrIdx - len, 'len': len },
            'condition': match[1]
        });
        const substr = query.slice(substrIdx);
        match = substr.match(ifRegex);
    }
```

Listing 5.6: Finding IF statements

After that, the if statements are looked at in reverse order. For each of the statements an `# ELSE #` and `# ENDIF #` tag are searched, beginning at the position of the `# IF #` statement. the `if_statements` array defined above is then updated with the position and length of the found ELSE and ENDIF tags. We now have all the information that is needed in order to evaluate the statements.

```
18      // find matching ELSE and ENDIFs
19      if_statements = if_statements.reverse();
20      for (let statement of if_statements) {
21          const start = statement['IF']['index'];
22          const endifMatch = query.slice(start).match(/#\sENDIF\s#/);
23          const elseMatch = query.slice(start).match(/#\sELSE\s#/);
24
25          if (elseMatch != null && elseMatch.index < endifMatch.index) {
26              const index = start + elseMatch.index;
27              const len = elseMatch[0].length;
28              statement['ELSE'] = { 'index': index, 'len': len };
29          }
30
31          const index = start + endifMatch.index;
32          const len = endifMatch[0].length;
33          statement['ENDIF'] = { 'index': index, 'len': len }
```

Listing 5.7: Finding matching ELSE and ENDIF

The condition of the IF statement needs to be evaluated and depending on whether it is satisfied or not, the IF or ELSE parts of the query need to be removed. This is done by slicing the query at the positions and lengths we just collected.

```
34          const satisfied = parseCond(
35              statement.condition, word, connectedLines, words);
36
37          let result = query.slice(0, statement['IF']['index']);
38          if (satisfied && statement["ELSE"] == undefined) {
39              // Add content between IF and ENDIF
40              result += query.slice(
41                  statement['IF']['index'] + statement['IF']['len'],
42                  statement['ENDIF']['index']
43              );
44          } else if (satisfied && statement["ELSE"] != undefined) {
45              // Add content between IF and ELSE
46              result += query.slice(
47                  statement['IF']['index'] + statement['IF']['len'],
48                  statement['ELSE']['index']
49              );
50          } else if (!satisfied && statement["ELSE"] != undefined) {
51              // Add content between ELSE and ENDIF
52              result += query.slice(
53                  statement['ELSE']['index'] + statement['ELSE']['len'],
54                  statement['ENDIF']['index']
55              );
56          }
57
58          result += query.slice(
59              statement['ENDIF']['index'] + statement['ENDIF']['len']
60          );
61          query = result;
62      }
63
64      return query
65  }
```

Listing 5.8: Removing parts depending on the condition

The conditions are evaluated by first splitting them at the operators and then evaluating each part recursively.

```
function parseCond(condition, word, connectedLines, words) {
    const ops = condition.match(/(.*)\s+(OR)\s+(.*)/)
        || condition.match(/(.*)\s(AND)\s+(.*)/);
    const negated = condition.startsWith("!");
    let satisfied = false;
    if (ops != null) {
        const lhs = parseCond(ops[1], word, connectedLines, words);
        const rhs = parseCond(ops[3], word, connectedLines, words);
        if (ops[2] == "OR") {
            satisfied = lhs || rhs;
        } else {
            satisfied = lhs && rhs;
        }
    } else if (negated) {
        satisfied = !parseCond(
            condition.slice(1), word, connectedLines, words);
    } else {
        if (condition == "CURRENT_WORD_EMPTY") {
            satisfied = (word.length == 0);
        } else if (condition == "CURRENT_SUBJECT_VARIABLE") {
            satisfied = (words.length > 0 && words[0].startsWith("?"))
        } else if (condition == "CURRENT_PREDICATE_VARIABLE") {
            satisfied = (words.length > 1 && words[1].startsWith("?"))
        } else if (condition == "CONNECTED_TRIPLES_EMPTY") {
            satisfied = (connectedLines.length == 0);
        }
    }
    return satisfied;
}
```

Listing 5.9: Evaluating conditions

5.2 Past approaches

While developing QLever UI, we went through several iterations of how to build the autocompletion queries. With each approach came some downsides which we tried to improve in the next iterations. The following examples will show our approaches by searching for object suggestions. Suggestions for subjects and predicates can be queried similarly.

5.2.1 The naive approach

Our first approach was quite simple, with no configuration possible at all. Suggestions were retrieved without their names and in no particular order. This was realized by a simple query which took the connected lines of the users query and completed the current line.

```
1 SELECT ?qlui_entity WHERE {
2    <connected_lines_here> .
3    <current_subject> <current_predicate> ?qlui_entity .
4    FILTER REGEX(?qlui_entity, "^<%CURRENT_WORD")
5 }
```

Listing 5.10: The naive approach: Suggesting objects

While this approach is easily implemented for predicate and object suggestions, it didn't work for subjects. Suggesting subjects would have introduced a query where each part of a triple is a variable, which is not supported by QLever.

We slightly improved this approach by adding the possibility of ordering the results. The ordering was done by adding a new setting called *scorePredicate* which was added to every autocompletion query in order to retrieve a score.

```
SELECT ?qlui_entity WHERE {
  <connected_lines_here> .
  <current_subject> <current_predicate> ?qlui_entity .
  ?qlui_entity <scorePredicate> ?qlui_score .
  FILTER REGEX(?qlui_entity, "^<%CURRENT_WORD")
}
ORDER BY DESC(?qlui_score)
```

Listing 5.11: Suggestions with ordering

5.2.2 Adding names to suggestions

Our second approach added three more settings to the back end: *subjectName*, *predicateName* and *objectName*. They each were allowed to contain any valid SPARQL code. It was not intended for them to contain complete SPARQL queries, but rather only those triples that would find the name for a given subject / predicate / object. It was required for them to use the variable `?qlui_name` for the entity name. Usage of these settings was as in the following query. The `<objectName_clause>` parts are to be replaced by the *objectName* setting. We also removed the previously introduced *scorePredicate* in favor of counting how often an entity occurred and ordering the result by this count.

```
SELECT ?qlui_object ?qlui_name ?qlui_count WHERE {
  {
    {  # Filter entities by their IRI and try to find their names
      SELECT ?qlui_object (COUNT(?qlui_object) AS ?qlui_count) WHERE {
        <connected_lines_here> .
        <current_subject> <current_predicate> ?qlui_entity .
      }
      GROUP BY ?qlui_object
      HAVING REGEX(?qlui_object, "^<%CURRENT_WORD")
    }
    OPTIONAL {
      <objectName_clause> .
    }
  } UNION {
    { # Filter entities by their names
      SELECT ?qlui_object (COUNT(?qlui_object) AS ?qlui_count) WHERE {
        <connected_lines_here> .
        <current_subject> <current_predicate> ?qlui_entity .
      }
      GROUP BY ?qlui_object
    }
    <objectName_clause> .
    FILTER REGEX(?qlui_name, '^\"" + word + "')
  }
}
ORDER BY DESC(?qlui_count)
```

Listing 5.12: Suggestions with names

We later also added three more settings for alternative names which could be configured in order to query synonyms. The usage of these settings in the autocompletion query is similar to the usage of `<objectName_clause>` above.

Although this approach was already quite advanced, it still had some problems. The largest part of the autocompletion queries was still hard-coded with no possibility to customize them on a per back end level. While they worked quite well on some back ends, they were still too simple for others. Also, the need for altering the query in some edge cases arose, which was not possible with this approach.

5.2.3 Fully configurable queries

We then decided to no longer pursue the notion of only partially configuring the autocompletion queries and instead having them fully configurable. In order for this to work, we came up with the template language that we already described previously. In this first iteration, there were only three settings. One for subject, predicate and object autocompletions each. While the autocompletion itself worked quite well now, one issue was that writing and configuring these queries became increasingly difficult. This was due to the fact that the queries became much larger. Because it was now possible to alter the query depending on several conditions using the `# IF #` statements, some parts of the query needed to be repeated across the same or even different queries.

The warm up queries and patterns were added in order to solve this issue. It was now possible to define some parts of the queries as warm up query or pattern and then referencing them in an autocompletion query using a placeholder.

6 Discussion

This chapter concludes our work for the QLever UI. It also shows a list of what improvements could be done to further refine our work.

6.1 Conclusion

We have discussed our user interface for QLever. It facilitates writing SPARQL queries even when a user has no extensive insight into a knowledge bases structure. The context-sensitive autocompletions and next word suggestions, together with the ability to handle synonyms, make it easy to write queries even when a user doesn't know how the entities in the knowledge base are named. We showed how QLever UI is interacted with from a normal user's perspective. We explained all the elements in the user interface and what they do, as well as how the features like the suggestions and name tool-tips work. We also discussed the admin interface and the most important configurations that need to be done in order for the suggestion engine to work as desired. Finally, we showed the template language we devised in order to build the autocompletion queries and how these queries are evaluated prior to sending them to the SPARQL back end.

6.2 Future Work

Lots of work went into the creation of our new QLever interface. Even though it already functions quite well and we have come a long way since our first concept for QLever UI, we are aware that not every part of it works perfectly and some improvements can be done.

6.2.1 Finding connected lines

Our algorithm for finding connected lines for the autocompletion queries works well as long as the query isn't too deeply nested, using sub queries or constructs like `UNION`. Our current solution mainly searches connected lines by searching for variables that have a relation to the variable(s) in the current line. This algorithm has no awareness for different name spaces and sub queries, which can lead to wrong suggestion when for example invoking them inside a sub query. A better algorithm could be devised that uses the information from the query parser in order to know exactly what kind of context it is located in and only use the lines that are related to this context. Since QLever UI already has a query parser that is actively used for other purposes, implementing such an algorithm should take less than a day. For more information on the query parser, see D. Kemens thesis about QLever UI [13].

6.2.2 Fully configurable name queries for name tool-tips

Our name tool-tips currently use the *subjectName*, *predicateName* and *objectName* settings that we originally introduced for one of our intermediate autocompletion approaches (see Section 5.2). This means that only a part of the query can be configured in the back end, while the rest is hard-coded. This solution works, but it is possible that some knowledge base might make it necessary to implement these queries using the same template language that is already used for the autocompletion

queries sometime in the future. Our template language should already be capable enough to also be usable for these queries. Also, since all the functions for parsing and evaluating such queries already exist, the main work for this problem would be in replacing the old query-building mechanism with the new one. This should also be doable in a few hours.

Bibliography

[1] H. Bast and B. Buchhold, "Qlever: A query engine for efficient sparql+ text search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 647–656, 2017.

[2] Wikimedia Foundation and Wikidata Contributors, "Wikidata main page." https://www.wikidata.org/wiki/Wikidata:Main_Page. [accessed 21-June-2021].

[3] Wikimedia Foundation and Wikidata Contributors, "Wikidata statistics page." https://www.wikidata.org/wiki/Wikidata:Statistics. [accessed 23-June-2021].

[4] S. Harris, A. Seaborne, and E. Prud'hommeaux, "Sparql 1.1 query language." https://www.w3.org/TR/sparql11-query/, 2013. [accessed 21-June-2021].

[5] G. Klyne, J. J. Carroll, and B. McBride, "Rdf 1.1 concepts and abstract syntax." https://www.w3.org/TR/rdf11-concepts/, 2014. [accessed 23-June-2021].

[6] RDF Working Group, "Resource description framework (rdf)." https://www.w3.org/RDF/, 2014. [accessed 23-June-2021].

[7] E. Prud'hommeaux and A. Seaborne, "Sparql query language for rdf." https://www.w3.org/TR/rdf-sparql-query/, 2007. [accessed 27-June-2021].

[8] S. Malyshev, J. Kress, and L. Pintscher, "Wikidata query service." `https://query.wikidata.org/`, 2015. [accessed 27-June-2021].

[9] H. Bast, F. Bäurle, B. Buchhold, and E. Haussmann, "Broccoli: Semantic full-text search at your fingertips," 2012.

[10] Triply B.V., "Yasgui." `https://yasgui.triply.cc/`. [accessed 4-September-2021].

[11] P.-Y. Vandenbussche and B. Vatant, "Linked open vocabularies." `https://lov.linkeddata.es/dataset/lov/`. [accessed 4-September-2021].

[12] S. Campinas, "Live sparql auto-completion," in *International Semantic Web Conference*, 2014.

[13] D. Kemen, "Qlever ui - building an interactive sparql editor to explore knowledge bases," 2021.