

Bachelor's Thesis

---

# QLever UI - Building an interactive SPARQL editor to explore knowledge bases

---

Daniel Kemen

Examiner: Prof. Dr. Hannah Bast

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

September 1<sup>st</sup>, 2021

**Writing Period**

16.06.2021 – 01.09.2021

**Examiner**

Prof. Dr. Hannah Bast

# 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.

Bad Krozingen, 07.09.2021

---

Place, Date



---

Signature



# Abstract

QLever (pronounced "clever") is an efficient full-text search engine (SPARQL + Text engine) that computes complex search queries on huge semantic knowledge bases and text corpora like Wikidata, OSM, and ClueWeb. The basic design of QLever was originally described in a paper for CIKM'17 by H. Bast and B. Buchhold[1].

Writing queries for specific information in very huge knowledge bases with billions of information requires prior knowledge about the contents and structure of the underlying data set. Therefore exploring huge data sets and writing comprehensive search queries can be complex and time-consuming.

Together with Julian Bürklin, I developed the interactive SPARQL editor "QLever UI" that provides an easy-to-use interface to write search queries for the QLever engine. QLever UI makes it easy to write syntactically correct search queries in the SPARQL language. Due to its ability to parse and evaluate even complex SPARQL search queries it can provide some very helpful features like context-sensitive and ordered auto-suggestions, syntax highlighting, name lookups, and an inline search that finds data attributes by synonyms of their names.

QLever UIs context-sensitive suggestions depend on the current cursor location inside the query and even evaluate partial and grossly incomplete parts of the query while the user is writing it. As the query becomes longer the suggestions become more relevant by evaluating all the already written parts of the query.

With QLever UI one can easily explore the scopes of a knowledge base with almost no background knowledge about the structure or contents of the dataset directly while writing the query itself.

The code is open source and can be found on GitHub (see chapter 5). The paper describes our design and usability approaches as well as the technical backgrounds of our work.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Semantic knowledge bases . . . . .	1
1.2	SPARQL and QLever . . . . .	4
1.3	Writing queries . . . . .	7
<b>2</b>	<b>Related Work</b>	<b>11</b>
<b>3</b>	<b>QLever UI Features</b>	<b>15</b>
3.1	Multiple highly configurable backends . . . . .	15
3.2	Syntax highlighting and structure . . . . .	22
3.3	Context-sensitive auto-completion . . . . .	23
3.4	Prefixes . . . . .	29
3.5	Shortcuts . . . . .	31
3.6	Tooltips . . . . .	32
3.7	Feedback on execution . . . . .	32
3.8	Sharing and saving . . . . .	33
3.9	Visualizing results . . . . .	34
<b>4</b>	<b>Behind the Scenes</b>	<b>37</b>
4.1	Parsing the query . . . . .	37
4.1.1	First approach - using only RegEx . . . . .	38
4.1.2	Second approach - using placeholders . . . . .	38
4.1.3	Third approach - using grammar parsers . . . . .	39

4.1.4	Lessons learned . . . . .	39
4.1.5	Current approach . . . . .	40
<b>5</b>	<b>How to use QLever UI</b>	<b>45</b>
<b>6</b>	<b>Conclusion</b>	<b>47</b>
	<b>Bibliography</b>	<b>51</b>



# List of Figures

1	The "Time in Space" example as simplified SPARQL . . . . .	5
2	Basic structure of a SPARQL query . . . . .	6
3	Basic structure of the PREFIX statement . . . . .	7
4	Screenshot of the whole QLever UI . . . . .	16
5	Administrator configuration page for backends . . . . .	20
6	Access, edit and add examples in QLever UI . . . . .	21
7	Selection and feedback on selected backend . . . . .	22
8	Dark theme of the query editor . . . . .	23
9	Simple keyword suggestion . . . . .	24
10	Context-sensitive suggestions . . . . .	25
11	Suggestions based on synonyms and tooltips . . . . .	27
12	Suggestions of variable names . . . . .	28
13	Suggestions of variables inside a SELECT clause . . . . .	29
14	Suggestions of variables inside a SELECT clause with grouped variable	30
15	Visualization of results . . . . .	35



# 1 Introduction

QLever UI is an easy-to-use web interface to write and execute search queries for the search engine QLever that adds plenty of useful features (see chapter 3) to make writing and understanding queries easier.

QLever uses the SPARQL query language to describe both the search criteria and the expected results[2] while QLever itself is the executing engine that actually searches for the results on huge knowledge bases. QLever was originally developed by H. Bast and B. Buchhold[1].

This thesis describes our approaches to build an interactive user interface for the QLever engine that supports different backends and knowledge bases and supports the user while writing queries and exploring the knowledge base.

Knowledge bases, SPARQL, QLever, and our QLever UI will be briefly introduced in the following.

## 1.1 Semantic knowledge bases

Semantic knowledge bases can store information and relations between them in a very simple and structured format which allows computers and software to easily work with this data. Other than in continuous texts in reference books the information in

Subject	Predicate	Object
The Hitchhiker's Guide to the Galaxy	publication date	1979-01-01
The Hitchhiker's Guide to the Galaxy	author	Douglas Adams
Douglas Adams	occupation	Novelist
Douglas Adams	date of birth	1952-03-11

**Table 1:** Sample of relational knowledge base entries

semantic knowledge bases follow well-defined formatting rules which are easy to read and interpret by algorithms.

One of the most common formats is RDF (*Resource Description Framework*) which basically stores information in a simple "Subject - Predicate - Object" triple format and is suggested by the "*World Wide Web Consortium*" W3C[3]. For example, RDF could contain various information formatted as shown in table 1.

As you can see a subject is followed by a predicate (the "attribute") and the predicate is followed by an object which is the actual value of the attribute. In this case, one of the information stored is that the "*author*" of "*The Hitchhiker's Guide to the Galaxy*" is "*Douglas Adams*".

Subjects and objects (so-called "entities") can overlap which means that "*Douglas Adams*" is not only the "object" of the "*author*" predicate for the book but can also be a separate subject in the knowledge base. "*Douglas Adams*" has his own predicates which contain objects that might have predicates as well - in this example the predicate "*occupation*" for "*Douglas Adams*" is given.

This allows modelling deeply nested dependencies and relations between entities. One can use these connections to access and link information which is very useful as shown in the following.

There are various knowledge bases for different use cases in existence. Since the open-source knowledge base "Freebase" (about 1.9 billion triples)[4] was acquired by Google and merged with Wikidata in 2015[5] the Wikidata knowledge base has

become one of the largest publicly available semantic knowledge bases for general knowledge. It contains billions of information about places, people, events, terms, objects, and many more.

However, there are many other use cases where storing information in a semantic format like this is useful (e.g. storing specific geographic or meteorological data). For example, QLever and QLever UI were tested and actively used with a knowledge base from OpenStreetMap which gives you access to their comprehensive geographical data[6].

For this thesis, we will use simple examples based on the Wikidata knowledge base since these examples will be more expressive for the purpose of demonstrating the QLever UI features.

In Wikidata and other RDF knowledge bases the information is not stored by their human-readable names but with internal IDs. For example, the book "*The Hitchhiker's Guide to the Galaxy*" in WikiData has the ID **Q3107329**. According to RDF, the IRI notation (*Internationalized Resource Identifier*, a generalization of *Uniform Resource Identifier* called URI)[7] is used. The full IRI of the book as it is stored within Wikidata is: `http://www.wikidata.org/entity/Q3107329`. The IRI for the "*author*" predicate is referred by `http://www.wikidata.org/prop/direct/P50`. Usually, there is only a reasonable small amount of prefixes used within the IRIs of a knowledge base.

Also, there are some literals inside WikiData (like dates and strings) where according to RDF the format consists of the lexical form and the formats IRI or language tag (for strings). For example, the *name* of *Douglas Adams* in the English language is stored as `"Douglas Adams"@en` and his *date of birth* is stored as:

`"1952-03-11T00:00:00"^^<http://www.w3.org/2001/XMLSchema#dateTime>` where `@en` marks the language of the string and `http://www.w3.org/2001/XMLSchema#dateTime` is the IRI of the date and time format.

## 1.2 SPARQL and QLever

Obviously requesting information from a knowledge base with many relations and information requires a very precise formulation of the question itself. It's relevant to specify which information should be returned and which search criteria and limiting rules should apply. For example, a "question" (called a query) could be:

**"What are the top 3 countries with the highest total amount of time spent in space by all female astronauts coming from that country?"**

*(The answer is the United States, Russia, and Italy by the time written. One can easily find out if this information is still correct using QLever UI, QLever, and the Wikidata dataset)*

As shown in this example one can use much-nested information together and even use grouping, aggregations, and ordering to request precise results.

In order to write these kinds of queries, the "SPARQL Protocol And RDF Query Language" is used. SPARQL 1.1 is an international recommendation published by the World Wide Web Consortium (W3C) in 2013[2]<sup>1</sup> and is a very comprehensive grammar for a query language.

By now only parts of the W3C recommendation are implemented in QLever<sup>2</sup> but this already gives a lot of possibilities while writing queries. QLever UI is extensible for upcoming features on QLever (see chapter 4).

While QLever UI helps writing correct SPARQL queries the QLever engine is evaluating those queries and actively searching for the correct solutions in the knowledge base. Neither the details of the SPARQL language definition nor the QLever implementation itself will be part of this paper.

---

<sup>1</sup>The initial version 1.0 was released in 2004

<sup>2</sup>The current version of QLever and its list of features is available on GitHub: <https://github.com/ad-freiburg/QLever>

```
SELECT DISTINCT ?country (SUM(?time_in_space) as ?sum_time)
WHERE {
    ?astronaut <occupation> <astronaut>
    ?astronaut <gender> <female>
    ?astronaut <citizenship> ?country
    ?astronaut <time-in-space> ?time_in_space
}
GROUP BY ?country
ORDER BY DESC(?sum_time)
LIMIT 3
```

**Figure 1:** The "Time in Space" example as simplified SPARQL

We will cover the basic structure of SPARQL and the problems one might have while writing a query in order to demonstrate how QLever UI can help with these problems.

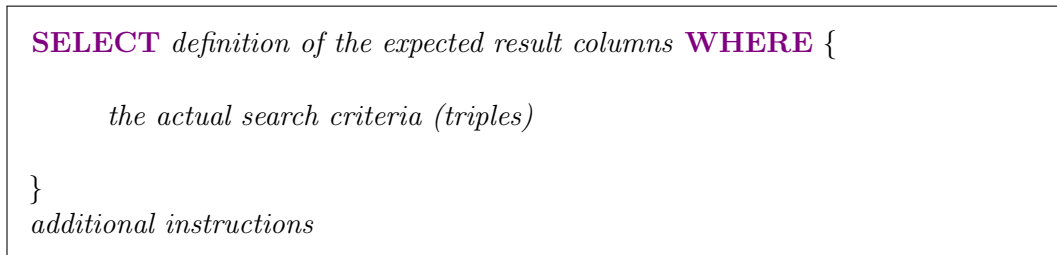
To visualize the basic concept of SPARQL figure 1 shows a (simplified) SPARQL version of the question above and will be used as a reference. For simplification subjects and predicates were replaced by their names instead of using the fully qualified name (IRI) as described in chapter 1.1.

As one can see the SPARQL query is fairly readable to humans thus requires some knowledge about the language itself. The language also uses the "Subject-Predicate-Object" triples as you can see within the brackets in lines three to six.

In addition, there are some language keywords (highlighted violet) that actually contain a signal for the search engine. Some variables (prefixed by a question mark and highlighted in blue) are used as placeholders for the actual data we want to retrieve.

The basic concept of a **SELECT** query is shown in figure 2.

The given additional keywords in figure 1 affect the outcome of the SPARQL query. The keyword "**DISTINCT**" in the first variable part for example tells the search



**Figure 2:** Basic structure of a SPARQL query

engine to leave out duplicate results while "**LIMIT**" tells the search engine how many results should be returned. The variables in the "**SELECT**" part of the query tell which columns (values) should be contained in the result. In this case, not all variables used in the query are used in the output because we are only interested in the countries and the sum of the time in space.

All keywords have specific places where they are allowed to occur in order to align with the SPARQL grammar[2]. Some other expressions have even more complex conditions. For example, the "**SUM**" keyword is only allowed in combination with variables that are also part of a "**GROUP BY**" statement within the same query (they need to be part of the aggregation in order to be summed up). The "**DISTINCT**" keyword needs to be the first keyword after the "**SELECT**" statement while the "**LIMIT**" keyword may only occur after the "**WHERE**" selection (the so-called *solution modifier*).

The triples used within the "**WHERE**" clause limit the output and must refer to IRIs as they are stored withing the RDF knowledge base.

SPARQL supports "**PREFIX**" statements to simplify a query. One can use "**PREFIX**" statements as shown in figure 3 instead of writing the full IRI. This makes reading and writing queries a lot easier due to the fact, that IRIs often share the same prefixes as described in chapter 1.1.

There is a detailed description of the different parts of a query and what elements



```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?politician WHERE {
    ?politician wdt:P106 wd:Q82955 .
}
additional instructions
```

**Figure 3:** Basic structure of the PREFIX statement

may occur inside the query on the W3C website[2]. This language definition also includes nesting of query parts, filters, optional clauses, and many more which are also supported by QLever and QLeverUI. In fact, you will see in chapter 4 that QLever UI uses (parts of) this definition directly to ensure a query is well-formatted and it does the right suggestions at the right place.

People that use QLeverUI might be more or less familiar with the SPARQL syntax and grammar which means that they might need different levels of support while building a correct query. Prior knowledge is definitely required in order to use QLever UI but it can help you find the right expressions in the right spots.

There are some more problems when writing SPARQL that simply require some prior knowledge or at least take some time for refactoring the query (prefixes for example). We will go into detail about this while discussing the QLever UI features in the next section as well as in chapter 3.

## 1.3 Writing queries

The SPARQL syntax itself is widely similar to the database query language "SQL"[8] and its grammar is mainly easy to read for humans but for actively writing and extending a query one needs to know the semantic rules and keywords as well as the structure of the underlying dataset and its contents.

In SQL one can query for rows and columns in a table inside your database[8]. In order to get the right values one needs to know which tables and columns are present within the database. One also needs to know their names and what they contain to be able to query for the correct results.

Depending on the size of the database it only takes a reasonable amount of time to explore the existing tables and fields and find the right place where the information one is looking for is located.

QLever does not search on databases but on knowledge bases that contain billions of information in one list of RDF triples. Going through all entries in order to see which information is stored inside the knowledge base will become very time-consuming even if one does this with the help of a search engine instead of going through it manually. Big knowledge bases have tens of billions of triples stored inside (see chapter 1.1).

When one looks for an entity in a knowledge base (let's say "*Alan Turing*") one simply can not know what IRI is assigned to this person and what information for *Alan Turing* is stored in the knowledge base. One must ask or search for IDs, names, predicates, and possible values (objects) manually, by using another search query or using some sort of external documentation on the knowledge base.

Searching for a specific attribute can be hard since they might have different names than one would expect. For example, the occupation of "*Alan Turing*" could be hidden in an attribute called "*job*", "*occupation*" or "*profession*" and one needs to know the right term in order to find the right predicate and its IRI.

Given a huge knowledge base (with IRIs and unknown IDs for almost everything) one could impossibly know "what to ask for" in order to get the expected results without doing further research about the structure and contents of the knowledge base.

Obviously not only the query mainly consists of IRIs and literals - the result set will do too. For getting human-readable results one must actively ask for human-readable values (like the name of an entity). Due to the fact that strings (e.g. names) might be present in different languages additional language filters might be required to get meaningful results which in most cases leads to larger queries.

Also one would need to manually manage prefixes or deal with the full IRIs within the query. When copying and editing already existing queries/examples one must look up each IRI to know what a triple in the query stands for in order to understand, extend, or manipulate the query.

It takes a lot of time and resources to find the correct expressions even for allegedly simple queries. Also, for the same reason, given queries and their results might be hard to read and understand afterward even when the SPARQL syntax/grammar is easy to understand.

In the following, I will present the user interface we created and how it solves these problems by providing a variety of features while writing SPARQL queries.



## 2 Related Work

The paper "QLever: A Query Engine for Efficient SPARQL+Text Search" by Hannah Bast and Björn Buchhold (2017)[1] describes the basic concepts of the QLever search engine and was published alongside a demo that was shipped with a simple web interface to send and execute queries to QLever. It already offered some features like clearing the cache and downloading the results as well as a simple visualization of the result while the input was a simple text area with no additional features.

QLever UI was built on top of this demo providing its additional features (see chapter 3) to make this simple interface to QLever a full-featured interface for not only working with the data but also working with and developing QLever and tools that make active use of it.

There are many other SPARQL / query builder interfaces present. Many interfaces try to actually remove the need for writing SPARQL or any query language at all by providing interfaces that rely on a custom interface of selectable options and individual input fields while having their very own visualization (e.g. MashQL[9] - similar features to SPARQL with a UI similar to Yahoo pipes, SPARKLIS[10] - an approach of building a natural language formulation of the query, or Broccoli[11] - a tree-shaped query formulation/visualization).

In the following only related work that actively supports writing SPARQL will be discussed since other approaches, while being relevant in the context of querying knowledge bases, vary a lot in their overall approach and goals.

The WikiMedia Foundation that maintains Wikipedia and WikiData offers its own service for sending SPARQL queries to WikiData called "WikiData Query Service"[12] that supports both: actually writing SPARQL as well as an interactive query builder interface. It uses some similar concepts to enrich the user's input and output by providing highlighting and output formatting, a comprehensive list of samples, and suggestions for already known variables. Its custom interface for adding prefixes and filters without actually writing them in the query visualizes the query while building the SPARQL query alongside. Suggestions while typing require prior knowledge (one must add and type a prefix in order to search an entity by name) and suggestions inside the query are not context-sensitive (suggesting even language keywords in the wrong spots). The query builder interface however considers already given filters. Hovering over the query parts is also supported and error reports are very precise by highlighting the corresponding lines. Overall the custom query builder interface is in focus prior to actively writing SPARQL.

Yasgui[13] is a SPARQL editor which also provides syntax highlighting and basic code editor features for SPARQL using the CodeMirror editor and its tokenizer/language mode. Suggestions are made by using the Linked Open Vocabularies service API for all properties and classes within the "**WHERE**" clause. This approach doesn't support context-awareness at all but can be extended as we will see in the following. Except for predicates and classes, only variables and prefixes are suggested. Constructs based upon the already written query parts are not supported and suggestions do not check for duplicates. Therefore Yasgui suggests adding the same prefix/variable/line multiple times. Incompleteness/errors are shown directly within the line they occur. One must enable the completions manually and they do not actively suggest using prefixes. Yasgui comes with a comprehensive set of visualization options like tables, charts, (3D) maps, timelines, and more. Sharing a query is supported as well as writing multiple queries in different tabs.

Stéphane Campina presented GoSparqled[14] which has a similar approach of context-

aware auto-completion as QLever UI and is built using the Yasgui interface adding the recommendation feature to Yasguis auto-completion plugin. The completions are context-aware using the already given parts of the query when generating suggestions. Real names/synonyms are not supported at all. For ranking the suggestions a fixed algorithm is proposed mainly built upon the assumption that the amount of occurrences is the best indicator for relevance.

Hannah Bast et. al. [15] published a paper on efficient SPARQL auto-completion mainly focusing on the performance and relevance aspects of context-sensitive auto-suggestions and also discussed different approaches to achieve relevance. Their approach was tested on a variety of different knowledge bases and describes approaches that can be implemented and used with QLever and the QLever UI leading to fast and relevant suggestions. The customizable settings per backend allow these kinds of optimizations.

Since QLever UI was developed together with Julian Bürklin there are more aspects of QLever UI (especially related to the communication with QLever itself and gathering the suggestions) described in this thesis "QLever UI: A context-sensitive user interface for QLever"[16].





## 3 QLever UI Features

The user interface of QLever UI is designed to allow immediate access to the knowledge base so the query editor is always in focus (see figure 4).

There are additional features and options as well as additional help located around the editor. Above the actual editor, one can find additional information and resources as well as an option to switch backends (see chapter 3.1). Below the editor, there are various buttons to get or manipulate results. Right below the actual results will be shown once the query is executed.

### 3.1 Multiple highly configurable backends

One can set up and manage different QLever instances within one QLever UI instance. This is a one-time configuration process per instance done by any registered administrator. Because of our comprehensive configuration options, QLever UI can adapt to very different setups. Since one can have different instances and especially different knowledge bases to work with, QLever UI is built to make working with multiple instances as easy as possible and runs separately from QLever itself.

For setting up and managing QLever backends the UI offers a password-protected user interface with many configuration options per backend. It can be accessed under the "resources" menu icon on top (see figure 4). The password protection was necessary to be able to provide a public service built on top of QLever UI without allowing any

QLever UI

Resources

Wikidata

Index Information
Backend Information
Shortcuts / Help

```

1 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 PREFIX wikibase: <http://wikiba.se/ontology#>
4 PREFIX schema: <http://schema.org/>
5 SELECT ?name ?pic ?sitelinks WHERE {
6   ?person wdt:P31 wd:Q5 .
7   ?person ^schema:about/wikibase:sitelinks ?sitelinks .
8   ?person wdt:P18 ?pic .
9   ?person schema:name ?name .
10  FILTER (lang(?name) = "en") .
11 }
12 ORDER BY DESC(?sitelinks)

```

Execute
Download
Share
Reset
Clear cache
Analysis
Examples

3. Context sensitive suggestions
Automatically add names to result
Clear the cache before every request

Query results:

791,255 lines found
51ms in total
50ms for computation
1ms for resolving and sending

Limited to 40 results. Show all 791,255 results.





	?name	?pic	?sitelinks
1	Barack Obama		309
2	Ronald Reagan		299
3	William Shakespeare		283
4	Albert Einstein		280

Figure 4: Screenshot of the whole QLever UI

regular user to alter the settings. It was preferred over a local configuration file on the server to allow easy and fast changes of any setting during the runtime and not presume access to the executing server. It also allowed us to sort configuration into different views and add detailed descriptions and additional features to each option to increase the ease of use.

The backend is built upon Python Django because the web framework already provides easy to extend features for authentication and storing information with multiple database adapters as well as an easy-to-use development server for staging or testing environments. Also, it easily adapts to different server environments and can live behind various webservers like Apache, Nginx, or Gunicorn<sup>1</sup>.

The overall goal was to achieve a high level of independence for QLever UI from the QLever backends and their infrastructure to adapt to very different setups as they are given in real world applications (see chapter 1).

All backend configurations can be exported and imported at any time which makes it easy to provide templates or pre-configured backends.

One may find details on the configuration option in J. Bürklin's thesis on QLever UI [16]. These multiple options for each backend make QLever UI as flexible as QLever by supporting knowledge bases with different structures and having the ability to highly customize the query handling and suggestions to the current needs.

For example suggestion ordering, language filtering, synonym queries, and many more features can be configured and adapted for the actual backend. Due to the fact that we designed QLever UI to actively use the actual QLever instance itself for providing most of its features the configuration directly correlates with the features we can provide. For example, the order of suggestions is not fixed and can be optimized by finding accurate clauses.

---

<sup>1</sup><https://docs.djangoproject.com/en/3.2/howto/deployment/>

We evaluated different approaches especially for the auto-completion (chapter 3.3) and naming feature (chapter 3.6) where QLever UI would make some predictions/-suggestions based on local dictionaries or services. We decided that external services as used by other SPARQL UIs (see chapter 2) would be an unwanted dependency and won't lead to relevant results.

We also evaluated an approach where QLever UI used its own pre-processing methods to gather prefixes, names, and synonyms from QLever or the knowledge base files itself. This approach required heavy and time-consuming pre-computation the had to be done for each backend independently. Thus it required enormous amounts of additional resources and storage in the QLever UI instance.

Additionally, it appeared that it was not as flexible as we needed it to be to adapt to different versions and instances of QLever and its knowledge bases.

Since QLever already provided features we could use and has become very fast in its procedures we decided not to keep any additional information and datasets inside the QLever UI database itself but to let QLever UI combine, create, send, and evaluate SPARQL queries in the background in order to gather the information needed. Using the individual configuration of each backend QLever UI now mainly decides on when and how background queries are executed and displayed in order to provide the best context- and backend-specific support.

Supported SPARQL keywords/features and a list of common prefixes are not provided by QLever right now so we decided to make their configuration a manual step. The approaches of pre-collecting these information were dropped due to the mentioned reasons and the assumption that in future versions QLever itself will return these values anyway.

This approach makes QLever UI itself very lightweight (it has very low requirements on disk space or computation power but uses QLever's resources) and removes the

need for any additional pre-computation or dependencies. QLever UI is ready to use right after setting it up (see chapter 5) and connecting a working QLever instance.

A paper of the Institute of Computer Science Hellas published on the IUI'97[17] describes an intelligent user interface like this: *„Intelligent user interfaces are characterized by their capability to adapt at run time and make several communication decisions concerning "what", "when", "why", and "how" to communicate [...] based on [...] determinants constituents, goals, and rules".*

The goal they wanted to achieve was a user interface that *"can be easily customized to the requirements of different application domains and users groups and can be re-used with minor modifications in different applications"*

Despite not completely following the suggested approach the QLever UI was designed with similar goals in mind being as flexible as possible and adapt to different domains (different knowledge bases/backends) and user groups (by customizable UI features) while mainly taking care of the communication process itself.

By being light-weight and separate from the data source and the computation plus having the additional authorization layer for configuration the QLever UI was designed to be an easy to setup „window" inside various knowledge bases and information being evaluated or provided by one or multiple providers in one interface. Features like sharing queries (chapter 3.8) complete this approach.

Since there could be different QLever instances with different knowledge bases we also needed to use different names or entities for filtering and searching inside the chosen backend (instance). This is also completely covered by our configurations (figure 5). We abstracted our queries as far as we could so that only the relevant, actually changing, parts of our queries need to be set in the configuration.

Also, one can manually add so called "warm up queries" per backend. These queries can be executed in the background in order to force QLever to load some

## Change backend

**General**

**Name:**

Choose a name for the backend that helps you to distinguish between multiple backends

**Slug:**

Name used in the URL of this backend; MUST only use valid URL characters (in particular, no space)

**Sort Key:**

Sort key, according to which backends are ordered lexicographically; DO NOT SHOW if this value is zero

**Base URL:**

The URL where to find / call the QLever backend (including http://)

☐ **Use as default**

Check if this should be the default backend for QLever UI

**Preprocessing**

**Source Path:**

Local (absolute or relative) path to the source .nt file QLever uses

**UI Suggestions**

**Default Maximum:**

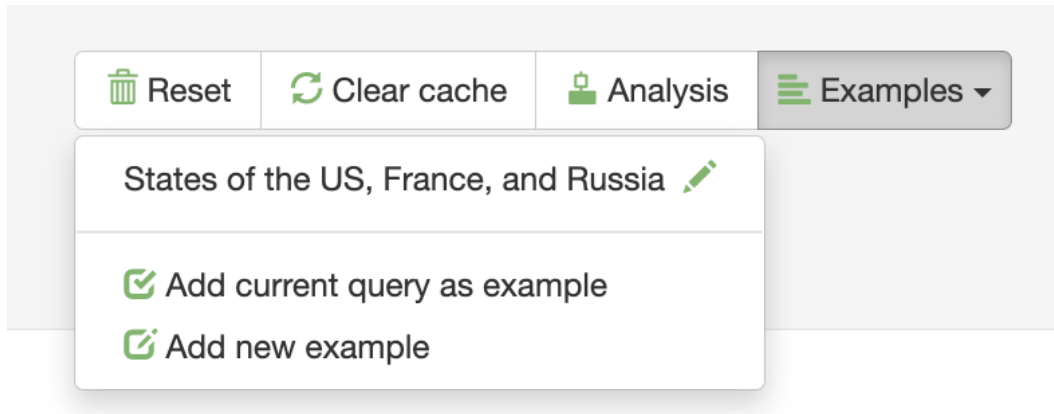
The default for how many lines are shown in the first request

**Figure 5:** Administrator configuration page for backends

information into its cache by requesting specific information (and allow some time for the execution).

Executing "warm up queries" is supported in the admin interface, via command line interface and via (authenticated) API call.

Additionally to the technical differences between multiple QLever instances the admin configuration also offers a feature to add backend-specific examples to QLever. One can either add examples in the admin interface or save the current query directly in



**Figure 6:** Access, edit and add examples in QLever UI

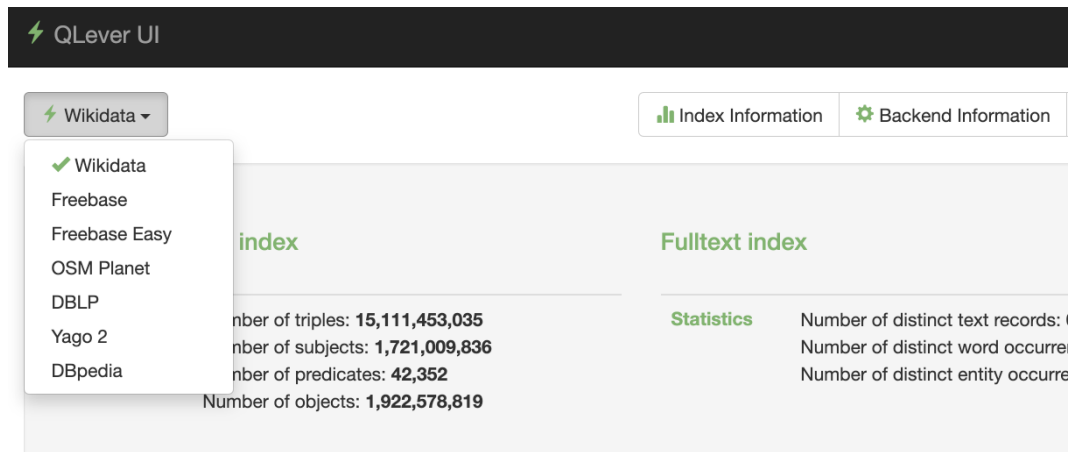
the regular interface (if logged in, see figure 6).

Examples can be accessed by everyone (not only logged-in users) and demonstrate the variety of possible queries for this backend. They usually give a good starting point to new users and allows them to keep and store different queries for testing or evaluating purposes. Of course, examples can also be changed, deleted, exported, and imported.

In the QLever UI interface, the currently active backend is always visible on the top left and can be changed with two clicks. The default backend on the first page load as well as the order of the available backends is subject to configuration. Afterward, the user's selection is stored in the user's session and taken on recurring visits.

Each backend has its own URL and can either be called directly or chosen from the list on the top left corner (figure 7). There is also an option for the existence of hidden backends that cannot be chosen without knowing their URL (e.g. for testing purposes).

QLever UI automatically connects to the current backend on page load and gives instant feedback if a backend is available at the moment. If the selected backend is available the interface gives additional information about the current index and



**Figure 7:** Selection and feedback on selected backend

its contents (as they are provided by QLever) and the backend configuration itself (figure 7).

## 3.2 Syntax highlighting and structure

QLever UI comes with a language parser for SPARQL syntax (see chapter 4) which allows the UI to highlight and outline specific language keywords and constructs (see figure 4). QLever UI marks language keyword (violet), variables (blue), prefixes (bold), strings (red), and delimiters (grey).

This helps to visualize different parts and recognized patterns of the query. If there is a typo or a missing character the missing or wrong highlighting helps to identify the issue very fast. Also, it improves readability and shows the semantic correlation of different parts of the query.

Only keywords and features that are supported by the active backend (according to the configuration) are highlighted (see chapter 3.1) so that QLever UI can adapt to older or newer versions of QLever.



```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 SELECT DISTINCT ?astronautLabel ?time_in_space WHERE {
5   ?astronaut wdt:P106 wd:Q11631 .
6   ?astronaut wdt:P2873 ?time_in_space .
7   ?astronaut rdfs:label ?astronautLabel .
8   FILTER (LANG(?astronautLabel) = "en") .
9 }
10 ORDER BY DESC(?time_in_space)

```

**Figure 8:** Dark theme of the query editor

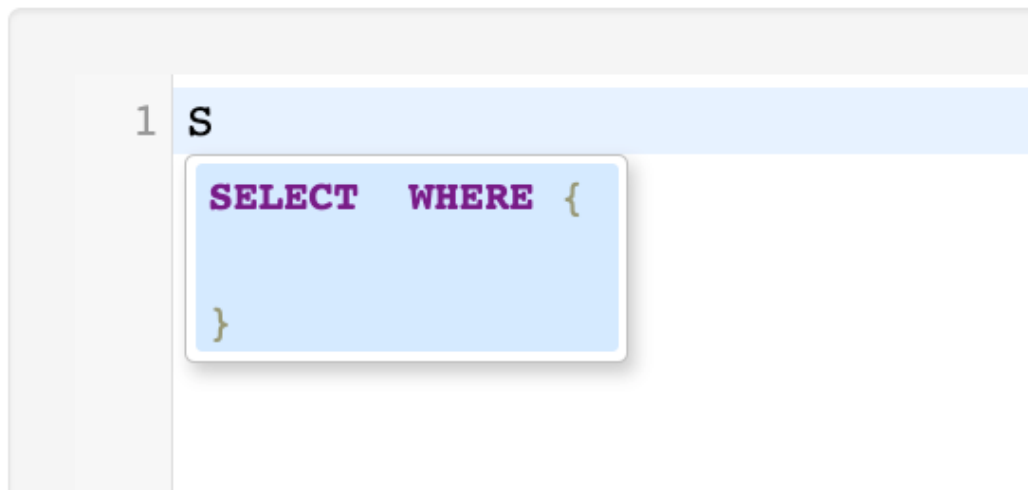
There is also a "dark theme" for QLever UI which changes the color scheme of the whole interface and the syntax highlighting. This enhances visual ergonomics and is considered to be more comfortable for those who prefer dark mode (figure 8).

The QLever UI text editor also provides simple helpers like line numbers for a better overview (and better reference) as well as it enforces users to use indentations and line breaks whenever they enhance the readability. Nested parts of the query are automatically indented by a configurable amount of spaces.

The indentation should ensure that the code is easy to read and understand and no one would mess up the query language. QLever itself doesn't require adding those line breaks but since some suggestions and hints in the editor are bound to the current line some features also benefit from a clean query structure. Enforcing the structure, therefore, fulfills different goals at once.

### 3.3 Context-sensitive auto-completion

QLever UI supports context-sensitive auto-completion and suggestions "as you type" for keywords, variables, constructs, and all entities and predicates. When the user starts typing a word QLever UI automatically suggests language keywords (syntax)



**Figure 9:** Simple keyword suggestion

but also names and other variable parts of a query based on the cursor position within the query.

When choosing a completion the corresponding query will be adjusted automatically and the cursor position moves to the next relevant gap to fill. It's not necessary to navigate by mouse or arrow keys at this point.

After adding the basic **SELECT** template by choosing the suggestions (figure 9) the cursor automatically jumps between the brackets since this is the next place in the query where to add custom query parameters.

This was an intended design choice because commonly starting with the "**WHERE**" clause is easier since you don't know which variables you are going to select before writing the "**WHERE**" clause. Pressing "**TAB**" simply jumps between all gaps in the query. This works even for nested queries.

Suggestions for language keywords are always based on the position inside of the query. As described in chapter 1.2 the "**DISTINCT**" keyword for example is only valid between "**SELECT**" and "**WHERE**" ("**SELECT**" clause) while for example, the "**OPTIONAL**" keyword is only valid between the brackets ("**WHERE**" clause).

```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 SELECT WHERE {
4   ?a wdt:P106 wd:Q11631 .
5   ?a
6 }

```

wdt:P450	"astronaut mission"@en
p:P450	"astronaut mission"@en
wdt:P166	"award received"@en
p:P166	"award received"@en
wdt:P734	"family name"@en
p:P734	"family name"@en
wdt:P108	"employer"@en
wdt:P2030	"NASA biographical ID"@en
p:P108	"employer"@en
p:P2030	"NASA biographical ID"@en
wdt:P410	"military rank"@en
p:P410	"military rank"@en
wdt:P361	"part of"@en

Figure 10: Context-sensitive suggestions

Also, the "**DISTINCT**" keyword is only valid exactly one time within a "**SELECT**" clause while the "**OPTIONAL**" clause could occur multiple times inside a SPARQL query. Following these semantic rules, QLever UI suggests those keywords only when they are valid. How this is done will be explained in chapter 4.

When the context-sensitive completion is turned on QLever automatically evaluates the given (partial) query in order to make suggestions based on the already given query parts.

In the example in figure 10 "wdt:P106" again stands for "*occupation*" and "wd:Q11631" represents the profession "*Astronaut*". So the variable ?a is now set to contain all entities in the knowledge base with the occupation "*Astronaut*". When adding another line and typing the variable ?a again the suggestions in figure 10 QLever UI suggests predicates like "*astronaut mission*" and "*NASA biographical ID*".

In this example, QLever UI has already evaluated that entities that are contained in

**?a** and therefore have the occupation "*Astronaut*" only have a subset of all available predicates and that predicates like "*astronaut mission*" occurs relatively often within the set of astronauts. As described in chapter 3.1 the ordering conditions are subject to the configuration but in this example simply using the number of occurrences already gives reasonable and useful results as you can see in figure 10. More details on the executed query that lead to these results see J. Bürklins thesis[16] and H. Basts paper on auto-completion[15].

In some scenarios requesting context-sensitive suggestions can take some time. QLever UI supports an option to send context-sensitive queries (slow) alongside with insensitive queries (fast) when collecting suggestions. After an individually defined amount of time QLever UI would in this case prefer showing the insensitive suggestions (that are almost always fast) over the sensitive ones to ensure the fast availability of suggestions at any time. QLever UI highlights sensitive and insensitive suggestions differently.

As already mentioned there is also support for synonyms that will be automatically used while typing. When typing "*job*" as done in figure 11 QLever UI automatically knows that "*job*" is a synonym for *occupation*. It automatically suggests using "*occupation*" and shows both names as well as the actual ID and IRI (with prefixes, see chapter 3.4) in its list of suggestions. We designed this to be as easy as possible for the user because this way the user can directly accept the suggestion and must not even notice that he or she used the wrong name to search for the attribute. This almost completely erases one of the main issues about writing queries as mentioned in chapter 1.2.

It is possible to configure a backend to use another or even multiple languages for these suggestions and synonyms. This might be useful if a user is not used to specific terms in a foreign language.

As described in chapter 3.3 QLever UI does not keep its own list of synonyms but takes

```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 SELECT WHERE {
4   ?a wdt:P31 wd:Q5 .
5   ?a job
6 }

```

**p:P106** "occupation"@en / "job"@en

**wdt:P106** "occupation"@en / "job"@en

**wdt:P2868** "subject has role"@en / "job title"@en

**p:P2868** "subject has role"@en / "job title"@en

**p:P1366** "replaced by"@en / "next job holder"@en

**wdt:P1366** "replaced by"@en / "next job holder"@en

**wdt:P1365** "replaces"@en / "previous job holder"@en

**p:P1365** "replaces"@en / "previous job holder"@en

Figure 11: Suggestions based on synonyms and tooltips

them directly from the knowledge base if present. In WikiData there is a predicate for names/labels and also one for alternative names (in different languages) which we make use of instead of keeping our own dictionaries or using external services.

In addition to keywords and context-sensitive entities, QLever UI also suggests smart variable names by using its knowledge about the real names of entities. In the example in figure 11 we asked for **P106** which QLever UI knows is named "*occupation*".

QLever UI automatically suggests speaking names for the new variable: "**?*occupation***" and even better "**?*person\_occupation***" as shown in figure 12. Although users are not required to use these suggestions this gives the ability to very quickly write code (by accepting suggestions in every position in the query) and while doing this automatically using speaking variable names which is not only good style but also very helpful to keep the query meaningful and understandable for everyone.

Of course, QLever UI can also suggest context-sensitive subjects and objects. For example, given the query about astronauts and asking about their place of birth leads

```
1 PREFIX p: <http://www.wikidata.org/prop/>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
4 SELECT WHERE {
5   ?person wdt:P31 wd:Q5 .
6   ?person p:P106
7 }
  ?occupation .
  ?person_occupation .
```

Figure 12: Suggestions of variable names

to a list of places where astronauts in the knowledge base are born. Depending on the configuration they are already sorted by the number of occurrences within entities with the occupation "*astronaut*" (returning cities where the most astronauts are born on top). When you continue typing only the cities with names or ids (partially) match the input will be shown.

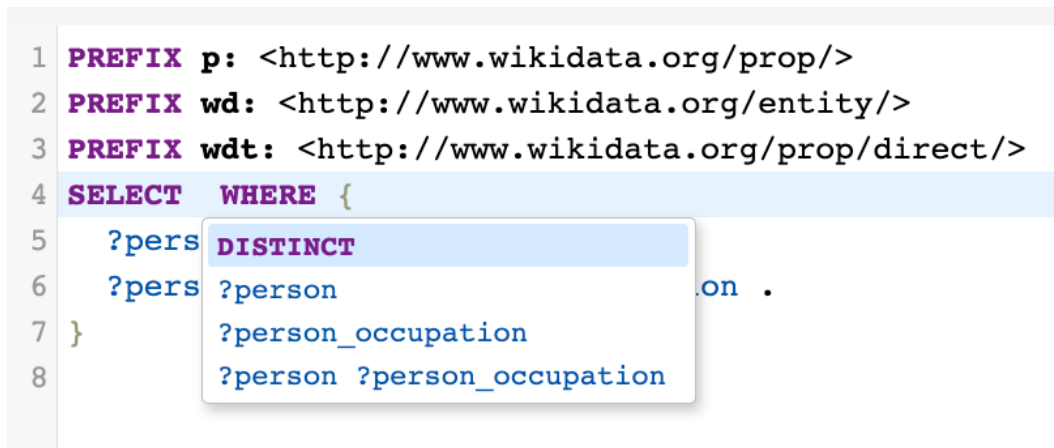
At any time QLever shows the number of suggestions found at the end of the line. This gives a hint on how limiting the existing query is by now and if there are still matching results left.

After accepting a suggestion for an object in the "**WHERE**" clause the UI automatically adds a new line below and jumps in this new line. In other parts of the query, this behavior adjusts to what is legitimate in this area (e.g. a white space as a delimiter). When starting the new line QLever UI will automatically suggest using one of the already defined and used variables as a start.

This means one can write a query by simply navigating through suggestions and will end up with a short, well-formatted, and easy-to-read query.

When jumping back to the "**SELECT**" clause (between "**SELECT**" and "**WHERE**") QLever UI automatically suggests adding variables too. It suggests every variable

```
1 PREFIX p: <http://www.wikidata.org/prop/>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
4 SELECT WHERE {
5   ?pers DISTINCT
6   ?pers ?person .
7 }
8
```



**Figure 13:** Suggestions of variables inside a SELECT clause

used within the query and even adding multiple variables at once (except for the already selected ones) in one step (see figure 13).

As mentioned in the instruction there are some more selections possible if a variable is grouped. So by simply adding the line "**GROUP BY ?person**" QLever UI ends up giving even more suggestions that became valid right after adding the aggregation to the query (see figure 14).

Again QLever UI suggests new meaningful names for these new aggregations by suggesting to add prefixes to the newly aggregated variables (e.g **?sum\_person\_occupation**).

### 3.4 Prefixes

As we discussed in chapter 1.1 the internal identifier for the predicate "*occupation*" in Wikidata is in fact "*http://www.wikidata.org/prop/statement/P106*" while for example the entity *computer scientist* is *http://www.wikidata.org/wiki/Q82594*.

Users need to either type the whole name including the prefix which leads to a very large and almost unreadable query or make use of the **PREFIX** statement by adding

```

1 PREFIX p: <http://www.wikidata.org/prop/>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
4 SELECT WHERE {
5   ?pers DISTINCT
6   ?pers ?person
7 }
8 GROUP BY ?person ?person_occupation
   (MIN(?person) as ?min_person)
   (MIN(?person_occupation) as ?min_person_occupation)
   (MIN(?person ?person_occupation ) as ?min_person ?person_occupation )
   (MAX(?person) as ?max_person)
   (MAX(?person_occupation) as ?max_person_occupation)
   (MAX(?person ?person_occupation ) as ?max_person ?person_occupation )
   (SUM(DISTINCT ?person) as ?sum_person)
   (SUM(?person) as ?sum_person)
   (SUM(DISTINCT ?person_occupation) as ?sum_person_occupation)

```

**Figure 14:** Suggestions of variables inside a SELECT clause with grouped variable

a new prefix above the **SELECT** statement. It's reasonable to assume that one would in most cases prefer using prefixes whenever possible.

That means whenever one needs an entity or predicate with a new prefix the prefix line needs to be added manually before continuing. As we have seen when discussing related work in chapter 2 the presence of prefixes is often required or completely ignored by other editors when it comes to suggestions.

QLever UI not only supports and highlights prefixes but also actively suggests using them and automatically fills them in if needed (even when copy and pasting a query inside QLever UI). When typing a name or entity QLever UIs suggestions automatically use the pre-configure prefixes to make the suggestions readable and short - even if this prefix was not used inside the query before.

If the suggestion is accepted by the user not only the shorted version is inserted but also the required "**PREFIX**" line above the "**SELECT**" query is automatically added to the query. This completely removes the actual need to care about prefixes at all (as long as a user doesn't want to make custom prefixes here) and lets QLever



UI organize the prefixes of the query automatically. Only required prefixes will be added once they are used for the first time in the query.

We decided to implicitly use and manage prefixes within QLever UI because the extraction of prefixes from names generally requires a lot of copy and paste and manually creation of repeating lines. This disturbs the user's workflow and is an unnecessary additional step. If a user wants to search for anything inside a specific namespace our search supports this too.

By now the **PREFIXES** need to be configured and named manually since QLever cannot be asked for a list of known prefixes with generated shorthands (see chapter 3.1).

## 3.5 Shortcuts

QLever UI supports a lot of shortcuts that are described directly in the user interface.

By hitting "**TAB**" one can easily jump between the relevant positions inside a query. QLever UI moves the cursor automatically between the "**SELECT**" clause, the "**WHERE**" clause, and the solution modifier at the end of the query. When using nested queries (like "**OPTIONAL**") the newly introduced positions will be used as well.

Hitting "**CTRL**" and "**ENTER**" will automatically send and evaluate the query and scroll down to the results table. "**CTRL + F**" and "**CTRL + R**" will open up a text search and text replace tool on the top-right corner.

Navigating and choosing suggestions works intuitively by using the arrow keys, "**ENTER**", and "**ESC**" or by simply scrolling and clicking in the suggestions.

For performance reasons not all possible suggestions in one spot are displayed by default but when the user scrolls through the suggestions the missing ones will be

loaded and appended to the list. Loading of additional entries is triggered when the user reaches the middle of the list to allow some time to load before the user reaches the end. The amount of appended suggestions increases in each step.

This makes using QLever UI extremely easy and fast to use because usually while writing a query no mouse (or switching between keyboard and mouse) is required if not preferred. Experienced computer users will be way faster by using the interfaces shortcuts.

## 3.6 Tooltips

As shown in figure 11 a SPARQL query uses the precise internal names and prefixes instead of their human-readable names that the UI suggested. This is because for the actual execution of the SPARQL query the entity must be specified with its full or prefixed IRI.

QLever UI helps to understand (even copied and pasted queries) by showing the canonical names of any entity in the query when hovering them with the mouse. If one is not using speaking variable names or is unable to keep track of each and every part and entity of his query this helps to turn the actual query back to a human-readable text.

## 3.7 Feedback on execution

QLever UI provides detailed logs and feedback on the executed queries and the internal steps while parsing the query in the console log of the web browser. When debugging or evaluating queries or backend configuration or the built-in log features helps to understand what happens internally. It can be turned on directly within the interface and differentiates various log categories.

Due to the fact that this QLever UI feature can not only be used while actively developing/extending QLever and QLever UI but is also very useful while configuring a new knowledge base, the logging feature is considered to be part of the UI instead of a separate development tool that would have been disabled in the production environment.

In addition to the actual query results and its own processing steps, the UI also shows the execution times and other relevant insights to the query execution as they are sent by QLever. This also includes the visualization of the detailed execution path of QLever for evaluation purposes. This is relevant if a query gets so complex that its execution time is highly dependent on the query structure.

### **3.8 Sharing and saving**

Once a query is executed QLever UI automatically generates a short link for the query and pastes it into your browser's URL. So whenever a query is executed one can simply copy the individual link (from the URL bar or the sharing window) and send it to someone else or link to this query in any document. The link is considered to be permanent and stored in the QLever UI instance. Only administrators can clear the query storage.

There is also a sharing window that can be opened within the user interface that provides additional links and commands prepared for easy copy and pasting them into other software or environments.

Using the sharing link is easier than copying and sending queries and very helpful while talking about the queries and results and collaborating with others.

### 3.9 Visualizing results

Having in mind that not only the development process of complex queries but also the actual results might be the focus for one of the common user groups the visualization of the results became a prominent feature of QLever UI.

Executing a SPARQL query in QLever UI is pretty simple. The interface directly sends it to the configured SPARQL backend once the "send" button is clicked or the already described shortcut (chapter 3.5) is triggered.

The results are shown in tables. They are automatically enhanced and cleaned instead of always showing the full qualified IRI for every result object.

For example, a decimal in the WikiData backend is stored as a literal (see 1.2):




```
"2110.0"^^<http://www.w3.org/2001/XMLSchema#decimal>
```

QLever automatically interprets these values and shows the canonical form without the format annotation (see 15).

Links and entities will get hyperlinked as well as images get loaded and displayed inline automatically. Some texts get truncated and language information is stripped automatically. This makes the results more readable. At any time the original results can be downloaded and will be shown by hovering over the corresponding cell.

By default output rows are limited automatically if the user hasn't specified a "**LIMIT**". This prevents the user from waiting for an enormous set of results when not explicitly defining a limit and improves performance. A small button to load more results, as well as the total number of results found, is added to the interface in this scenario.

Figure 15 shows a sample of the QLever UIs visualization of the results and execution information.

Query results:			34 lines found	67ms in total	66ms for computation	1ms for resolving and sending
	?cover	?person	?publication_date			
1	SPOTLIGHT	Spotlight	2015-09-03T00:00:00			
2	THE BIG SHORT	The Big Short	2015-11-12T00:00:00			
3		Carol	2015-05-17T00:00:00			
4		Mr. Smith Goes to Washington	1939-10-19T00:00:00			
5	AMERICAN PSYCHO	American Psycho	2000-01-21T00:00:00			
6		On Body and Soul	2017-02-10T00:00:00			

**Figure 15:** Visualization of results

For geographical results, a link to QLevers "MAP UI" will be automatically generated and shown in order to visualize the results properly.

Since the user groups of QLever UI and their intentions might be very different a download button of the untouched results without any modification is also present and helpful / required for further using the results gathered.



## 4 Behind the Scenes

An integral part of this thesis is the **README.md** that can be found in the QLever UI project repository<sup>1</sup> on GitHub.

It describes further details on the actual implementation and also contains a guide on how to extend the QLever UI tokenizer, parser, and the actual suggestions on a code level.

We preferred the **README** markdown documentation over explaining the details in the thesis due to the fact that it is easier to directly link code and relate to changes. Also for further development, the **README.md** can be adjusted and extended easily when QLever and QLever UI are extended in the future whilst this thesis won't get updated.

The general theoretical concepts of our implementation of the parser and our considerations of different approaches will be explained in the following.

### 4.1 Parsing the query

The main problem when parsing SPARQL queries "as you type" is that it's hard to always determine the correct location inside an incomplete query. This requires

---

<sup>1</sup><https://github.com/ad-freiburg/qllever-ui>

the parser to deal with partial words and constructs while being able to generate meaningful suggestions at almost every position inside the incomplete query.

During evaluation and testing, we implemented different methods of SPARQL parsing.

#### **4.1.1 First approach - using only RegEx**

It turned out that the initial "naive" approach, parsing queries using mainly regular expressions to search for keywords and specific spots inside the query, was prone to failure. The main problem is the possible incompleteness of one or many constructs (e.g. missing brackets or values) within the query which leads to wrong detection and wrong matches.

Our first approach had multiple issues when users typed something wrong or relevant parts that were an integral part of the pattern were missing. We also noticed some situations where we simply could not express everything we needed using only regular expressions due to the missing context sensitivity in RegEx itself. Furthermore, the regular expressions became highly complex and hard to extend very soon.

#### **4.1.2 Second approach - using placeholders**

Another approach was to simply use pre-defined templates and suggestions that contain placeholders to overcome the incompleteness. Whenever a user has chosen a construct that was incomplete we added placeholders on the given position in order to detect the incompleteness and skip the line or raise an error when there are placeholders left while executing the query.

This approach led to other problems for example when copy and pasting a query (or parts of a query) in and out or when users decide to not follow the predefined flow.



We minimized the false matches on our first approach while adding new boundaries for the user. Limiting users on free typing was considered an unwanted limitation.

#### **4.1.3 Third approach - using grammar parsers**

There are many implementations of parsers for the SPARQL grammar according to the W3C specification present. Using a full-featured parser would have easily provided the ability to parse the whole set of available SPARQL features without any need to manually configure them. The fact that QLever doesn't support all SPARQL features and even more adds custom features for full-text search required to adjust the grammar accordingly.

The main problem with this approach was that the SPARQL grammar and its parsers are only valid on complete queries that are syntactically correct. Due to the fact that we needed to deal with incompleteness and we needed to evaluate parts of those incomplete queries (when adding lines in the middle) this approach didn't work out either - especially when dealing with complex suggestions and connected query lines.

#### **4.1.4 Lessons learned**

Generally, we encountered issues when removing boundaries, limitations, or guidance from the user. Enforcing a specific order or limiting the input to only suggested/valid options in a given position makes it easier to parse the inputs and provide meaningful suggestions. The requirement to be as flexible as possible on the other hand makes this problem harder.

The fact that errors or incompleteness in a single line or position leads to irritations for parsing the whole rest of the query was encountered to be the biggest problem of our previous approaches.

We needed an approach that makes parsing on smaller local parts of the query while already generating a structured representation of the full query itself.

#### **4.1.5 Current approach**

In order to resolve the issues of our previous approaches, QLever UI now uses a tree-like representation which is parsed using a combination of an iterative walk-through and regular expressions to detect locations and constructs inside the query.

First of all, we are extending a simple tokenizer that it is provided by the code editor CodeMirror. It provides the separation of different lines, words, and characters while going through the editors content character by character. This basic separation of simple characters in lines, chars, and tokens already gives an easy way to interact with local parts of the query.

We extended this tokenizer to detect and differentiate even more parts of the query. This tokenization is used for syntax highlighting as well as for the following context and keyword detection. It distinguishes keywords, variables, literals, brackets, and more.

This leads to a local detection of single words or characters by naming the general type of the expression. The tokenizer is not meant to detect combinations, constructs, or parts of the query but being able to differentiate keywords, variables, literals, and delimiters helps a lot when it comes to this step.

The editor already splits all of its content into a DOM representation of the typed text which allows CSS styling reading the content line by line. It also allowed us to put information and highlights (e.g. the number of found suggestions or error signs) directly next to the corresponding line, to make selections and replacements and many more (see chapter 5).

We also bind features as name tooltips (chapter 3) on this representation using the standard event listeners on the relevant parts of the query. All these features solely rely on this first step on tokenization.

Next, we use the grammar definition of the W3C recommendation for representing the different parts in a query in an abstract format (our very own tree representation). The example below shows a possible tree and the corresponding positions of each part in the query as it is also logged by the UI itself.

- » PrefixDecl [0 to 195]
- » SelectClause [201 to 226]
- » WhereClause [232 to 512]
  - » SelectClause [247 to 279]
  - » WhereClause [285 to 370]
  - » SolutionModifier [371 to 394]
    - » GroupCondition [384 to 391]
- » SolutionModifier [513 to 536]
  - » OrderCondition [522 to 536]

Other than a regular SPARQL parser (and our first approach) we test for indicators and not for a complete match with the definition. For this, we mainly look at what introduces a new part of the query by going through the text until the beginning of a new part is detected.

Each part of the query (context) has different properties that correspond to the rules of the W3C recommendation and our custom formatting rules.

There is an object for each supported context that holds the definition and properties of every context a query could have (a complete list of a contexts properties can be found in the **README.md** documentation as mentioned in chapter 5).

When QLever itself adds new features one could simply add the newly supported context to this list. Each context has a name and some properties that tell what is legitimate inside this context and how to detect it. These could be single words as indicators or more complex regular expressions.

Additionally, there is an internal data structure for the suggestion constructs (referred to as "complex types"). Each construct has a readable name and additional parameters:

- a definition (how it is detected)
- a list of contexts it is compatible with
- a callback function that gathers the suggestions for this construct
- additional controlling parameters for UI features (auto-completion, line ends,...)

This definition is similar to the context definition and simply defines how to detect this construct inside the query and where to suggest it. The allowed scope of context is used to only detect and suggest the construct in the correct context.

Most important the individual callback function for each construct is used to combine all static and dynamic suggestions and generate a list of all available suggestions (see chapter 3).

The static suggestions are based on the grammar definitions of the construct (e.g. valid keywords or the ability to use variable names in this construct). Also the additional features QLever brings in for the full-text search are added by fixed methods (e.g. suggesting useful combinations) in the callback.

On the other hand, the dynamic suggestions are gathered by combining the valid part of the already written query and the configured parameters (e.g. additional filters, the ordering parameter for suggestions, and the lines to request the human-readable names). Those suggestions will be combined and considered for suggestions

by the editor (see J. Bürklins thesis on QLever UI [16] for more details on dynamic suggestions).

By default, the results are limited because right at the beginning there might be thousands of suggestions leading to long waiting times and a DOM overload. We implemented infinite scrolling that just recalls the query with other offset and limit parameters (increasing the limit on every reload) and adding the additional results to the lists.

Additional parameters in the construct definition control whether the construct is allowed to occur more than once, whether suggestions should be shown in every situation or only after the user typed in a matching word, and other requirements and implications of this construct that should be taken into account. As well as the context definition this list is also extensible if the supported language scope is extended.

Further explanation linking the files and codes responsible for the desired function is available in the **README.md** document of the repository.



## 5 How to use QLever UI

To run a QLever UI instance yourself you can simply download or clone QLever UI from our public GitHub repository:

**<https://github.com/ad-freiburg/qllever-ui>**

A detailed setup guide for QLever UI as well as further information on how to customize and extend QLever UI can be found in the **README.md** file in the repository.

By the time of writing, we also offer a Docker image of QLever UI which is also explained in the setup guide as well as a default configuration file to import a sample backend to get started quickly.

There is also a public demo available on:

**<https://qllever.cs.uni-freiburg.de/>**

QLever UI is distributed under an open source license so anyone is allowed to create forks or pull requests and further extend or customize the described features.





## 6 Conclusion

QLever UI addresses different user groups and needs by being highly adaptive to different use cases, QLever backends, preferences, and knowledge bases. Once a backend is configured QLever UI can be an easy-to-use interactive helper to query information from a huge knowledge base without having prior knowledge of the underlying data structures. This gives access to data that might be hard to reach by manually going through the process of discovering.

Writing SPARQL queries with QLever UI is fast and convenient while resulting in clean and well formatted queries.

We can achieve this by combining features known from code editors (e.g. suggestions, shortcuts, ...) with convenient features known from other end-user interfaces (e.g. examples, sharing, visualization of results, ...). The core of QLever UI is the context-sensitive auto-suggestion feature that is fast and makes active use of the knowledge base itself for all of its features including the support for synonyms.

Additionally QLever UI can provide useful tools for those who manage/setup or otherwise work with a knowledge base or actively develop on their own QLever instance caring more about the instance and its features than of the actual contents of the knowledge base.

In comparison with other SPARQL editors QLever UI doesn't introduce individual new interfaces but supports the process of actively writing pure SPARQL with

respect to the context of the actual position in the query at all times on a level we have not seen in other solutions yet. It supports a wide range of SPARQL features and simplifies the process of writing by not only supporting but actively using and suggesting prefixes, speaking names, constructs and many more.

Due to the ability to configure so many aspects of QLever UI it becomes a very useful tool for many fields of application and it's features can be optimized for each of them without touching the code.

The fact that QLever UI makes active use of the QLever engine without further need for additional computation power or space or any dependency on external services makes it flexible and lightweight which are valuable properties of a user interface.

The SPARQL grammar supports way more constructs than QLever UI and QLever by now but the QLever engine is actively developed and extended. Also, new fields and knowledge bases (like geographical or meteorological data) became relevant during the time of development.

These circumstances often forced us to reconsider choices and rebuild constructs because they no longer worked for the upcoming challenges. As a result, we created the configuration backend (chapter 3.1) and the query representations (described in chapter 4) that allow customization and extension of the supported constructs and features and adaption to the specific needs without changing the underlying logic.

# Bibliography

- [1] H. Bast and B. Buchhold, “Qlever: A query engine for efficient sparql+text search,” *CIKM’17*, 2017.
- [2] W3C, “Sparql 1.1 overview,” W3C recommendation, W3C, Mar. 2013. <https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>.
- [3] D. Wood, M. Lanthaler, and R. Cyganiak, “RDF 1.1 concepts and abstract syntax,” W3C recommendation, W3C, Feb. 2014. <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [4] Google, “Firebase API (deprecated).” <https://developers.google.com/firebase/>, 2021. [Online; accessed 19-July-2021].
- [5] K. G. team at Google, 2014. <https://plus.google.com/109936836907132434202/posts/bu3z2wVqcQc>, accessed 10-July-2021.
- [6] A. Ballatore, M. Bertolotto, and D. Wilson, “Geographic knowledge extraction and semantic similarity in openstreetmap,” *Knowledge and Information Systems*, vol. 37, 10 2013.
- [7] M. Duerst and M. Suignard, “Internationalized resource identifiers (iris),” RFC 3987, RFC Editor, January 2005. <http://www.rfc-editor.org/rfc/rfc3987.txt>.

- [8] *ISO/IEC 9075, Information technology - Database languages - SQL - international standard*. Internat. Inst. for Standardization, 1992.
- [9] M. Jarrar and M. Dikaiakos, “A query formulation language for the data web,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, pp. 783–798, 05 2012.
- [10] S. Ferré, “Sparklis: An expressive query builder for sparql endpoints with guidance in natural language,” *Semantic Web*, vol. 8, pp. 405–418, 2017.
- [11] H. Bast, F. Bährle, B. Buchhold, and E. Haussmann, “Broccoli: Semantic full-text search at your fingertips,” 2012.
- [12] W. D. Jens Ohlig, “A gentle introduction to the wikidata query service,” 2021. [https://www.wikidata.org/wiki/Wikidata:SPARQL\\_query\\_service/A\\_gentle\\_introduction\\_to\\_the\\_Wikidata\\_Query\\_Service](https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/A_gentle_introduction_to_the_Wikidata_Query_Service), accessed 18-June-2021.
- [13] L. Rietveld and R. Hoekstra, “Yasgui: Not just another sparql client,” in *The Semantic Web: ESWC 2013 Satellite Events* (P. Cimiano, M. Fernández, V. Lopez, S. Schlobach, and J. Völker, eds.), (Berlin, Heidelberg), pp. 78–86, Springer Berlin Heidelberg, 2013.
- [14] S. Campina, “Live sparql auto-completion,” *ISWC Posters & Demos / CEUR Workshop Proceedings*, vol. 1272, p. 477–480, 2014. [http://ceur-ws.org/Vol-1272/paper\\_157.pdf](http://ceur-ws.org/Vol-1272/paper_157.pdf).
- [15] H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle, “Efficient SPARQL Autocompletion via SPARQL,” 2021.
- [16] J. Bürklin, “Qlever ui: A context-sensitive user interface for qlever,” 2021.

- [17] C. Stephanidis, C. Karagiannidis, and A. Koumpis, “Decision making in intelligent user interfaces,” in *Proceedings of the 2nd International Conference on Intelligent User Interfaces*, IUI '97, (New York, NY, USA), p. 195–202, Association for Computing Machinery, 1997.

