University of Freiburg

Department of Computer Science

Bachelor of Science Thesis

Efficient and Effective Search on Wikidata Using the QLever Engine

presented by Johannes Kalmbach

Matriculation Number: 4012473

Supervisor:
Prof. Dr. Hannah Bast

October 3, 2018

Contents

Abstract

The QLever engine allows efficient semantic search on a combination of RDF knowledge bases and fulltext corpora using an extension to the SPARQL query language. In this thesis we present several improvements to QLever that allow performing almost arbitrary SPARQL queries on the Wikidata knowledge base. Those improvements heavily reduce the RAM usage of QLever without significantly increasing the time needed for query processing.

1 Introduction

Knowledge bases are an effective way of storing structured data in the form of triples. Each triple consists of a subject, a predicate and an object. E.g. the information which cities are capitals of a country can be stored in the following way:

Subject	Predicate	Object
`<Germany>`	`<capital>`	`<Berlin>`
`<France>`	`<capital>`	`<Paris>`
`<Colombia>`	`<capital>`	`<Bogota>`

The format for such knowledge bases is standardized in the Resource Description Framework (RDF). To obtain information from such knowledge bases the SPARQL query language can be used. For example we can retrieve the capital of Colombia from our initial example using the following SPARQL query:

```
SELECT ?capital_of_colombia WHERE {
  <Colombia> <capital> ?capital_of_colombia
}
```

However much of the knowledge in the world does not exist in such a structured form but as (human-readable) fulltext like in the following example from the Wikipedia article about the city of Bogotá (`https://en.wikipedia.org/wiki/Bogota`):

> "Important landmarks and tourist stops in Bogotá include the botanical garden José Celestino Mutis, the Quinta de Bolívar, the national observatory, the planetarium, Maloka, the Colpatria observation point, the observation point of La Calera, the monument of the American flags, and La Candelaria (the historical district of the city)."

One way to connect this (unstructured) full-text data to the structured data from a knowledge base is to annotate snippets of the text with entities from the knowledge base. In our example this would mean that we specify that the word "Bogotá" in our text refers to the entity `<Bogota>` in the knowledge base. In 2017 Hannah Bast and Björn Buchhold ([1]) presented an extension to the SPARQL language to perform queries on a combination of a knowledge base and an annotated text corpus. Using this extension we can find a list of all capitals that have a botanical garden:

```
SELECT ?capital WHERE {
  ?country <capital> ?capital .
  ?text ql:contains-entity ?capital .
  ?text ql:contains-word "botanical garden"
}
```

Bast and Buchhold also presented the QLever engine that is able to efficiently execute such SPARQL + Text queries. They showed that their system outperformed other state-of-the-art SPARQL engines with SPARQL + Text queries as well as with pure SPARQL queries.[1]
In the original QLever evaluation the Freebase knowledge base was used. Since this dataset has been shutdown in 2014 it is desirable to use and evaluate the QLever engine with Wikidata which has taken Freebase's place as the biggest publicly available general-purpose knowledge base. When trying to set up QLever with this knowledge base the following observation was made: QLever scales well with an increased number of triples in the knowledge base but the RAM usage drastically increases if the number of distinct entities becomes larger.

1.1 Problem Definition

The QLever engine originally was not able to load a complete RDF dump of the Wikidata knowledge base since the creation of the index (a preprocessing step that has to be performed before being able to execute queries) ran out of memory even on machines with more than 200GB. In this thesis we will introduce improvements for the QLever engine that allow us to build the index and run queries using less than 32GB of RAM without slowing down the execution of queries. We will also discuss mechanisms to effectively create queries for Wikidata.

1.2 Structure

In chapters 2 through 4 we will introduce the necessary background for our work. This includes

- The RDF format that is used to store knowledge bases.
- The SPARQL query language that is used to retrieve data from such knowledge bases.
- The Wikidata knowledge base.
- The QLever engine for SPARQL + Text queries and its extension to the SPARQL standard.

Chapters 5 through 7 describe several internal mechanisms of the QLever engine and how we reduced their RAM usage to make QLever work with the full Wikidata knowledge base. Chapter 8 describes an efficient implementation of SPARQL language filters that are an important feature for queries on large knowledge bases like Wikidata. In chapter 9 we will move our focus from the efficient execution of queries to their effective creation: We will describe some difficulties that arise when creating queries for Wikidata or other large knowledge bases. We will introduce the *Entity Finder*, an efficient search engine that can be used to find the internal representation of an entity in Wikidata given its (human-readable) name.

[1] Since the other engines had no native support for SPARQL + Text queries this feature had to be simulated e.g. by inserting the information of the text corpus into the knowledge base and adapting the queries.

2 SPARQL and the RDF Data Model

In this chapter we will introduce the RDF standard for knowledge bases and the SPARQL query language. We will limit ourselves to the basic features that are required for understanding the rest of this thesis. For completeness we link to the corresponding standardization documents.

The *Resource Description Framework* (RDF) is a data model with a well-defined semantic used to store and exchange information or knowledge in the form of *statements* that are made about (named) *entities*. It was standardized by the W3 consortium, the current version RDF 1.1 was published as a W3 recommendation in 2014 (all the standard documents can be found at `https://www.w3.org/RDF/`). RDF was originally intended to store the meta data of web resources like the author or the last modification date of a website. However it is also widely used to store more general information. RDF statements have the form of *triples*. Each triple consists of a *subject*, a *predicate* and an *object*. In the following the term *knowledge base* will stand for a set of RDF triples. For example the information that Johann Sebastian Bach was born in Eisenach, a German city, in 1685 could be expressed by the following knowledge base K_{Bach}:

```
<Johann Sebastian Bach> <place of birth> <Eisenach> .
<Johann Sebastian Bach> <year of birth> "1685" .
<Eisenach> <is a> <city> .
<Eisenach> <contained in> <Germany> .
```

The format we just used is (a simplified version of) the *N-Triples* format that is basically a list of triples that are separated by dots. Below we will introduce this format in further detail. A set of RDF statements can also be interpreted as a directed graph where the subjects and objects are nodes in the graph and each statement or triple is an edge from the subject to the object of the statement that is labeled with the predicate (see figure 1). In the following we will focus on the

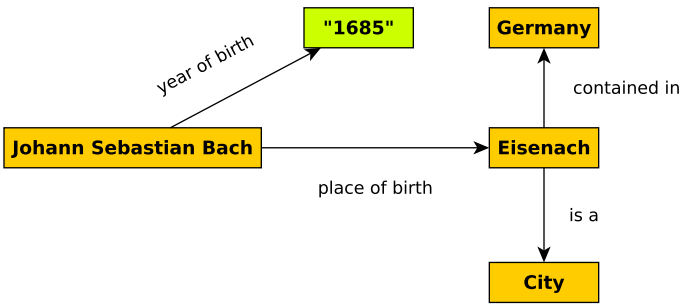

Figure 1: The knowledge base K_{Bach} in its graph representation

triple-based formats for RDF knowledge bases because the QLever engine currently only works with knowledge bases that are represented using triples.

2.1 The N-Triples Format

As already stated above, the *N-Triples* format is a relatively simple format to represent RDF statements. Its full specification can be found at `https://www.w3.org/TR/n-triples/`. The

N-Triples format consists of triples separated by dots. Within the triples *entities* are surrounded by angles (e.g. `<Eisenach>`) while value literals like `"1685"` are surrounded by quotation marks. Literals can only occur as the object of triples, the subject and the predicate of each triple must be an entity.

Using URIs to Indentify Entities

In real-life knowledge bases the entities are represented by *Uniform Resource Identifiers* (URIs). URIs have the same format as URLs but do not necessarily point to a web resource. For example the city of Leipzig in eastern Germany is represented by the URI `<http://www.wikidata.org/entity/Q2079>` in Wikidata. Using URIs makes it possible to uniquely identify entities across different knowledge bases because one can (or should) only introduce new URIs if one owns the corresponding URL namespace. For example if the team behind QLever would like to state its favourite place in Cologne we could use Wikidata's URI for Cologne but since there is no predicate that maps a city to the QLever team's favourite site of this city we would have to introduce a new URI from an URL space that we own. So our triple could for example look as follows:

```
<http://www.wikidata.org/entity/Q2079>
<http://qlever.informatik.uni-freiburg.de/favourite-place>
<http://www.wikidata.org/entity/Q4176> .
```

(`<http://...Q4176>` represents the famous Cathedral of Cologne in Wikidata). Because we reused the Wikidata URIs one can be sure that it is valid to search the Wikidata knowledge base for more statements about `<http://www.wikidata.org/entity/Q4176>` if one wants to find more information about this place which we have stated as our favourite. In this thesis we will not always use fully qualified URIs but sometimes use shortened entity names like `<Cologne>` or `<name>` for the sake of brevity and readability.

Language Tags and Datatypes

Value literals can be extended by a language tag or a datatype, this can be seen in the following example knowledge base:

```
<Cologne> <name> "Cologne"@en .
<Cologne> <name> "Köln"@de .
<Cologne> <population> "+1075935"^^<Decimal> .
```

The language tags (`@en` and `@de` in the example) connect a literal to a specific language (The city of Cologne is known under different names in German and English). Datatypes are entities themselves and are connected to a literal using `^^`. In our example `"+1075935"^^<Decimal>` tells us that the string "+1075935" should be interpreted as the decimal number 1075935.

2.2 The Turtle Format

The turtle format (`https://www.w3.org/TR/turtle/`) is an alternative way of storing RDF statements. It is an extension of the N-Triples format (every valid N-Triples file is also a valid Turtle file) but allows several ways to compress information that occurs repeatedly in many triples.

Prefix Directives

In section 2.1 we have seen that entity names should be fully-qualified URIs. Because of this many entities in a typical knowledge base share a common beginning. For example many entities in Wikidata start with `<http://www.wikidata.org/entities/`. The turtle format allows us to introduce a short name for this prefix using a *prefix directive*. So instead of the fully qualified triple

```
<http://www.wikidata.org/entity/Q42>
<http://www.wikidata.org/prop/direct/P31>
<http://www.wikidata.org/entity/Q5> .
```

("Douglas Adams is a human") we can write

```
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix wdt: <http://www.wikidata.org/prop/direct/> .
wd:Q42 wdt:P31 wd:Q5 .
```

which is much more readable and also saves a lot of characters when the short form of the prefixes is used in more triples.

Repeating Subjects and Objects

Typically many triples in a knowledge base have the same subject. When a Turtle triple is finished with a semicolon instead of a dot this means that we will reuse the subject of the previous triple and only specify a predicate and an object for the next triple:

```
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix wdt: <http://www.wikidata.org/prop/direct/> .
wd:Q42 wdt:P31 wd:Q5 ;  # Douglas Adams is a human
       wdt:P21 wd:Q6581097 ;   # ... gender male
       wdt:P106 wd:Q214917 .   # ... occupation playwright
```

When we want to repeat the subject and the predicate and only state a new object, the triples are ended with a comma:

```
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix wdt: <http://www.wikidata.org/prop/direct/> .
wd:Q42 wdt:P800 wd:Q25169 ,  # Douglas Adams notable work The Hitchhiker's guide ...
                wd:Q902712,  # ... Dirk Gently's Holistic Detective Agency
                wd:Q7758404  # ... The Private Life of Genghis Khan.
```

2.3 The SPARQL Query Language

To efficiently retrieve information from RDF knowledge bases the SPARQL query language has been specified by the W3 consortium (`https://www.w3.org/TR/rdf-sparql-query/`). Its syntax resembles the SQL language for relational data bases in several aspects like the used keywords. The most basic construct is a `SELECT` clause as we have already seen earlier in this thesis:

```
SELECT ?a WHERE {
  ?a <is-a> <Astronaut>
}
```

Formally this query returns the subjects of all triples that have `<is-a>` as a predicate and `<Astronaut>` as an object. Every token that starts with a question mark (`?a`) is a variable. Multiple query triples can be connected by a dot:

```
SELECT ?h ?date WHERE {
  ?h <is-a> <Human> .
  ?h <date_of_birth> ?date
}
```

The semantic of the dot is the logical "and" or more precisely a join on the columns where the triples have the same variable. So this query returns all pairs (`h, d`) such that `h` is a human and `d` is the date of birth of the same human (according to the semantics of single triples in the knowledge base).
Similar to the Turtle format we can also specify prefixes to make queries shorter and more readable:

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?human WHERE {
  ?human wdt:P31 wd:Q5  # <is-a> <human>
}
```

This internally expands the shorthand notation `wd:Q5` to `<http://www.wikidata.org/entity/Q5>`.

Sorting

It is often desired to sort the results of a query in certain way. This function is provided by the `ORDER BY`-clause of SPARQL:

```
SELECT ?h ?date WHERE {
  ?h <is-a> <Human> .
  ?h <date_of_birth> ?date
}
ORDER BY DESC(?date)
```

This query will sort the result such that the youngest humans (the ones most recently born) will appear first. Changing the clause to `ORDER BY ASC(?date)` would reverse the order. The SPARQL standard ensures that the ordering of different data types behaves as expected: Strings are sorted lexicographically while dates and numeric literals are sorted according to the values they represent. It is also possible to sort by multiple columns. E.g. the clause `ORDER BY DESC(?date) ASC(?h)` would sort by the date first (in descending order). Humans with the same date of birth would appear in alphabetical order.

Filtering

Another commonly used feature in SPARQL is the `FILTER` clause. Often we are only interested in results from a certain range:

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema>
SELECT ?h ?date WHERE {
  ?h <is-a> <Human> .
  ?h <date_of_birth> ?date .
  FILTER (?date >= "1950-01-01T00:00:00"^^xsd:dateTime)
}
```

This query only returns humans that were born in 1950 or later (`xsd:dateTime` is the canonical data type for date and time values used in all common knowledge bases).
Another important type of filter is the language filter. Literals only pass this filter if they have a language tag that meets the requirements of the filter. Language filters are described in detail in chapter 8 of this thesis. The SPARQL standards also specifies other types of filters. Those are not relevant for this work and are described in the SPARQL specification.

Advanced Features

The SPARQL language also contains many more features that are not of relevance for this work. Those include for example subqueries (queries whose results are then used in another query), aggregates like `GROUP BY` and the union of the result of multiple (sub-)queries. Those mostly behave like their equivalents in the SQL language.

3 The Wikidata Knowledge Base

Wikidata (`https://www.wikidata.org/`) is a free knowledge base that is maintained and hosted by the Wikimedia foundation which also maintains the famous Wikipedia and all of its sister projects. One of Wikidata's main goals is to centrally store the structured data from all the Wikimedia projects to avoid duplications. For example most of the data contained in the info boxes of Wikipedia articles about a country does not have to be maintained separately by the Wikipedia project but is dynamically loaded from Wikidata. In addition to this "internal" use by the Wikimedia foundation the Wikidata project also provides external APIs to access its information (see section 3.2).

3.1 Structure

In this section we introduce the format Wikidata uses to represent knowledge as RDF triples. We only describe aspects that are relevant for the rest of this thesis. A detailed description of Wikidata's RDF format can be found at `https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format`.

3.1.1 Fully-Qualified Statements

Wikidata does not only state simple facts or claims but also annotates them with meta data. For example Wikidata does not only state the fact that the German capital Berlin is populated by about 6.2 million people but also contains information about the source or reference of the statement, the date it refers to, etc. To represent this concept of meta data for statements in the RDF standard the Wikidata project uses intermediate nodes that are used to connect a statement to its value and its meta data. This can be seen in figure 2.

Berlin	population	abstract statement
wd:Q64	p:P1082	wds:Q64-D9D0[...]
Direct and Indirect Value for Statement		
wds:Q64-D9D0[...]	ps:P1082	"3611222"
wds:Q64-D9D0[...]	psv:P1082	wdv:8afdc[...]
Reference/ Source for Statement		
wds:Q64-D9D0[...]	wasDerivedFrom	wdref:d32d[...]
Direct and Indirect Date Qualifier		
wds:Q64-D9D0[...]	pq:P585	"2017-11-30"
wds:Q64-D9D0[...]	pqv:P585	wdv:659df57[...]
...	...	...

Figure 2: A Wikidata statement about the population of Berlin and some of its meta data. Long entity names are abbreviated. The abstract statement entity *wds:Q64-D...* represents one statement about Berlin's population. This abstract entity is then used as a subject in additional triples to connect the statement to its actual value, to its reference and to its date. Note that there are two triples for the value: One directly states the value "3611222". The other one introduces a new abstract entity *wdv:8a...* that is used to add additional meta data to the value in a similar way. Same goes for the date qualifier.

With this concept of meta data for statements it is now possible to have multiple statements about the population of Berlin for example for different points in time or if different sources state different numbers (it is hard to determine the exact population count of a big city).
When looking at figure 2 we observe that Wikidata contains multiple "flavors" of the same predicate: The id `P1082` always stands for the concept of the population of an entity. From this id we get the predicate `p:P1082` that maps an entity to a statement (a `wds:...` node), the predicate `ps:P1082` that maps a Wikidata statement to its value and many more. Another important variant is `wdt:P1082` which will be described in the following section. The expanded forms of those prefixes can be seen in Attachment A.

3.1.2 Truthy Statements

On the one hand the fully-qualified statements we just described contain a lot of information that is needed e.g. when using Wikidata as a source for scientific research. On the other hand this format also introduces a lot of overhead. In our example one might simply want to answer the question "What is the population of Berlin?" which for the format just introduced roughly translates to "What is the value of the most recent statement about the population of Berlin". For such simple queries Wikidata introduced the concept of *truthy* statements. For example the

most recent population statement of a city is truthy while older statements are not.[2] This is done by assigning each statement a rank. All statements with the highest rank are considered to be truthy. Note that there can be multiple truthy statements for the same combination of subject and predicate depending on the type of the predicate: When we state the population of a city we probably expect only one statement to be truthy. When we state which books were written by a certain writer we probably expect all the books by that author to be *truthy*. For truthy statements Wikidata contains triples that map the subject and the predicate directly to the value (see figure 3).

Berlin	population	3611222
wd:Q64	wdt:P1082	"3611222"

Figure 3: Wikidata triple for a truthy/direct statement on the population of Berlin

3.2 Obtaining Data

The Wikidata project provides several ways to access its data: Each item has a webpage where the most important facts about it can be viewed (for example `https://www.wikidata.org/wiki/Q64` for the city of Berlin (Q64)). For structured access to the data there is the *Wikidata Query Service*, (`https://query.wikidata.org/`) a SPARQL endpoint to the Wikidata project. It is also possible to download full dumps of the Wikidata knowledge base in the Turtle format (see section 2.2). This makes Wikidata interesting for this work since it allows us to set up an instance of the QLever engine with the full Wikidata knowledge base which was the initial motivation for this thesis. All the different ways to access Wikidata and the formats used there are described at `https://www.wikidata.org/wiki/Wikidata:Database_download`.

3.3 Statistics and Size

The Wikidata project claims to contain ca. 550 million statements on ca. 50 million entities.[3] However these numbers do not include the intermediate entities and triples used to add meta data to a statement. The actual number of triples in an RDF dump of Wikidata is interesting for us because this is the format QLever works on. It can be seen, among some other statistics, in the following table:

Number of Triples	7.1 Billion
Number of Literals	418 Million
Number of Entities	862 Million

Table 1: Statistics on Wikidata. Uses the Turtle dump of the full knowledge base from October 1, 2018. Every token that is not a literal is counted as an entity.

[2] The actual method of determining truthy statements is a little bit more involved but this abstraction suffices for the sake of this thesis.

[3] `https://www.wikidata.org/wiki/Wikidata:Statistics`, October 1, 2018

4 The QLever Engine

QLever is a combined SPARQL + Text engine that is being developed at the Chair of Algorithms and Data Structures at the University of Freiburg. In addition to standard SPARQL queries it supports a custom extension of SPARQL that allows combined search on a RDF knowledge base and a full text corpus when the entities of the knowledge base are linked to words in the text corpus. A detailed description of this extension and of QLever's internals can be found in [1]. When we refer to the *original version of QLever* in this thesis we always refer to the features described in that paper. In this thesis we will focus on the standard SPARQL part of QLever.

Technical Details And Terminology

The QLever engine is implemented in C++ and currently only supports the Linux OS on 64-bit machines. It can be run inside a Docker container. In QLever the complete knowledge base has to be imported at once. It does not support insertion or deletion of triples. This limitation allows it to optimize the internal data structures of QLever for fast query execution. Those data structures are called the *index* of a knowledge base and the process where a knowledge base is imported into QLever is called *index building*. Thus when we want to use a given knowledge base with QLever we first have to build the index for this knowledge base. Then we can start the QLever engine with this index and execute SPARQL queries.

Obtaining and Using QLever

The QLever engine is open-source and licensed under the Apache License 2.0. Its source code can be obtained from `https://github.com/ad-freiburg/QLever`. QLever contains a small web server that provides a simple user interface and a HTTP-GET API to perform queries. Some demo instances running QLever on various knowledge bases can be accessed via `http://qlever.informatik.uni-freiburg.de/`. Those instances are maintained by the QLever developers at the chair of Algorithms and Data Structures in Freiburg.

5 The Vocabulary of QLever

As we have seen before, RDF knowledge bases and SPARQL queries contain tokens like `"Douglas Adams"@en`, `wdt:Q42`, ... that form triples. The Qlever engine internally maps each of those tokens to a unique id. This is done by sorting all the tokens from the knowledge base lexicographically. The position of a token after this sort becomes its id (see figure 4 for an example). That way the lexicographical order of the tokens is the same as the numerical order of the ids. Before sorting the vocabulary QLever transforms numerical literals and date literals to a special representation such that the lexicographical order of the tokens reflects the order of the values the literals represent (e.g. "10" < "9" in lexicographical order but $10 > 9$ when interpreted as numbers).
We refer to this sorted set of all tokens from a knowledge base as its *Vocabulary*. The actual query processing inside QLever is done in the numeric id space which is way more efficient that working on strings.

```
<alan_turing>   <is_a>  <scientist> .
<ada_lovelace>  <is_a>  <scientist>
```

Id	token
0	<ada_lovelace>
1	<alan_turing>
2	<is_a>
3	<scientist>

Figure 4: A tiny knowledge base (left) and its sorted vocabulary (right)

5.1 Resolving Tokens and Ids

Since SPARQL queries and their results contain tokens from the knowledge base and since the query processing of QLever is handled in the id space one important task for the engine when executing a query is to translate between the token space and the id space: As a first step after parsing the query all tokens from the query that correspond to a token in the knowledge base have to be translated to their numeric id. Then the query is processed in the id space. After the ids of all possible results have been computed, they have to be translated back to the token space to obtain the actual result of the query. Consider for example the following SPARQL query:

```
SELECT ?city WHERE {
  ?city <is-a> <city> .
  ?city <contained-in> <Poland> .
}
```

It is intuitively clear that the QLever engine has to map the tokens `<is-a>`, `<city>`, `<contained-in>` and `<Poland>` to their corresponding ids. If executing this query yields for example 3, 7, and 9 as id values for the selected variable `?city`, the engine as a last step has to translate these back to string tokens like `<Warsaw>`, `<Krakow>` and `<Mielec>`. Since the Vocabulary of the knowledge base is stored in lexicographical order we can perform those mappings as follows: To convert an id to the corresponding token we can just look up the array at the position indicated by the id ($\mathcal{O}(1)$). To translate a token to the correct id we have to perform binary search on the array ($\mathcal{O}(\log n)$ where n is the size of the Vocabulary). This approach also allows us to find a lower or upper bound for the id of a given token. This allows us to efficiently perform SPARQL FILTER clauses (see section 2.3) directly in the id space. When we for example want to filter a certain result variable for tokens that start with the letter `'a'` we have to find a lower bound for the id of all words that lexicographically are greater or equal to the string `"a"` and an upper bound for all tokens that are lexicographically smaller than the string `"b"`. The binary search approach allows us to also find these bounds if one of those strings is not contained in the vocabulary. It would theoretically be possible to improve the runtime of the string-to-id translation to $\mathcal{O}(1)$ by additionally creating a hash table that maps strings to ids. The main disadvantage of this method is that it only allows us to perform the translation in one direction and that it consumes additional memory. Additionally the translation from tokens to ids is mostly not that time critical: Normally there are relatively few tokens to resolve within a query and thus the $\log n$ binary search in practice never becomes a bottleneck.

5.2 Improvements for Large Vocabularies

In the last section we have seen that the general approach for storing the sorted Vocabulary in QLever is using a contiguous array in RAM. In practice this approach works well for small to medium-size knowledge bases with a relatively small vocabulary that can be fit into RAM at once. However big knowledge bases like Wikidata or Freebase contain a lot of different tokens: For example the vocabulary of the full Wikidata needs about 80GB and contains about 1.2 billion tokens. In the following section we will introduce several strategies that can be used to deal with such large Vocabularies without completeley storing them in RAM.

5.2.1 Omitting Triples and Tokens From the Knowledge Base

The probably easiest way to reduce the RAM usage of the vocabulary and of the whole QLever index is to reduce the size of the knowledge base by deleting triples that are not relevant for the specific use case. Of course this is not a valid solution when our actual goal is to build a SPARQL engine that can deal with large-scale knowledge bases. In practice however we often know what we will use the knowledge base for and thus can prune the set of triples to fit our needs. When we for example want to build a QLever index based on Wikidata that can help historians with their research we can probably omit many triples from Wikidata that contain information about chemical elements (Wikidata contains a lot of biological and chemical knowledge). And if we know which languages are relevant for our users we can omit all triples that contain object literals with language tags that probably will never be searched for.

5.2.2 Externalizing Tokens to Disk

In most large-scale knowledge bases there are types of tokens that are used relatively seldom but contribute in a great amount to the total size of the vocabulary. These can for example be

- Literals in "exotic" languages. This is especially true for Wikidata because its concept is not to be biased towards any language.
- Very long literals like descriptions of entities.
- Abstract entity URIs that represent intermediate nodes and typically never appear in a query or a result. This is for example true for the `wds:`, `wdv:`, ... entities in Wikidata (see section 3.1)

The original version of QLever already contained the following mechanism to externalize certain literals:

- During the index build, decide for each literal whether it shall be externalized or not (this decision could for instance be made depending on the language tag of the literal or on its length).
- Prepend all literals that are to be externalized with a special, non-printable character that will cause them to lexicographically appear after all literals that are not externalized (the implementation uses a byte with the value 127. This works because in the N-Triples format that QLever uses internally all Tokens either start with `<` or `"` [see section 2.1]. The ASCII value of both characters is less than 127).

- After sorting the Vocabulary (see section 5.4 for the details of this procedure) write all the literals that start with the special externalization character to a separate file.

When starting the Qlever engine we only read the not-externalized literals into RAM. We can tell by the id of a token whether it was externalized or not because the ids of all externalized tokens are higher than the ids of all the tokens that were loaded to RAM. When resolving a string to its id, we first check whether this string meets the criteria for externalization. If this check is true, we resolve it using binary search on the externalized vocabulary on hard disk[4]. If not, we perform binary search on the array of "internal" words in RAM as described above.
We can see that this method is still able to correctly resolve ids to strings and vice versa. However, this method breaks the lexicographical ordering of the vocabulary under certain conditions: When sorting a set of ids that partly belongs to the internal and to the external vocabulary the internal ids will always appear before the external ids which does not necessarily reflect the lexicographic ordering of the corresponding tokens (see figure 5). Similar problems arise when filtering query results. In practice this is mostly not problematic since we aim to only externalize tokens that we mostly won't use.

Internal (RAM)	
Id	**token**
0	"Douglas Adams"@en
1	rdfs:label
2	wd:Q42

External (disk)	
Id	**token**
3	"Adams Duglas"@vep
4	wds:Q42-...-D97778CBBEF9

Figure 5: Internal and External Vocabulary for some tokens from Wikidata. Note that the id of *"Douglas Adams"@en* is lower than that of *"Adams Duglas"@vep* because the latter is externalized. Thus when both of them appear in a query result (e.g. when we do not filter for a specific language) their ordering is incorrect.

Contributions

The original version of QLever only supported the externalization of literals depending on their length and their language tag. We have extended this mechanism to also externalize entities if they match a certain pattern. For example in the Wikidata knowledge base we externalize all intermediate entities that only refer to abstract statement, value or reference nodes (see section 3.1).

5.3 Compression

For this thesis we have also evaluated possible memory savings when applying compression algorithms to the vocabulary. This is described in detail in chapter 6.

5.4 RAM-efficient Vocabulary Creation

Until now we have seen that once we have created our Vocabulary and also a suitable representation of the knowledge base in the id space we are able to perform the query execution in the id space.

[4]The externalized vocabulary is implemented using a data structure that allows us to perform binary search on an array of strings on disk. This still runs in $\mathcal{O}(n)$ but with the overhead of random accesses to the hard disk. The implementation details of this data structure are not relevant for this thesis and thus are omitted.

In this section we will put our focus on creating these data structures. Namely we need to create the bijective mapping from each token in the knowledge base to its lexicographical position within the vocabulary. In a second step we read the complete knowledge base and convert each triple of tokens to a triple of ids.[5] On a high level this can be done as follows:

1. Read all triples from the knowledge base and create the set S of all tokens (we need a set since tokens can occur in multiple triples).
2. Sort S according to the lexicographical ordering of strings.
3. Assign the id 0 to the first element of S, 1 to the second element, etc.
4. In a second pass over all the triples we can now map the whole knowledge base to the id space and store it efficiently using $3 \cdot sizeof(Id)$ bytes per triple.

5.4.1 Original Implementation

The original version of QLever implemented these steps in the following way

1. Read all triples and insert all tokens into a hash set (this ensures that each token is added to the vocabulary only once).
2. Store all the elements of the set in a (dynamic) array and sort this array.
3. The index of a token in the array now equals its id.
4. Transform the sorted array into a hash table that maps tokens to their ids.
5. During the second pass, read each triple and query the hash map for the corresponding id.

We can see that this algorithm is asymptotically optimal, since insertion and access to hash maps and hash sets run in $\mathcal{O}(1)$ per token in average. Thus steps 1 and 5 run in in $\mathcal{O}(t)$ if t is the number of triples in the knowledge base. Steps 2 (first half) and 4 run in $\mathcal{O}(n)$ where n is the number of distinct tokens ($n \leq t$). The sorting in step 2 is done in $\mathcal{O}(n \log n)$. The overall runtime of $\mathcal{O}(t + n \log n)$ is optimal because we have to access each triple of the knowledge base and because we have to sort the vocabulary.
The method just described performed well enough in practice for the original use case of QLever where the focus was put more on combined SPARQL + Text queries with huge text corpora but with relatively small knowledge bases. However, when trying to build a QLever index for the complete Wikidata knowledge base we ran into trouble for the following reasons:

- The whole vocabulary was kept in memory at the same time. (Only after the index build the most memory-consuming literals could be externalized to disk, see section 5.2.2)
- During the conversions from the hash set to the dynamic array and from the dynamic array to the hash table actually two copies of the vocabulary were kept in memory.

[5] This step is needed to create the *permutations*, the internal representation of the knowledge base QLever uses. They are described in section 7.1

- The chosen hash set and hash table implementations (dense_hash_map and dense_hash_set from google::sparsehash[6]) have a relatively big memory overhead per value since they are optimized for speed.

For those reasons it was not possible to build the Wikidata index even on machines with 128GB of RAM because we ran out of memory during the creation of the vocabulary. To deal with this problem, some minor improvements immediately come to mind when reading the list of problems just stated:

- Choose a hash set and hash table implementation that is optimized for a low memory overhead (e.g. the sparse_hash_set from google::sparsehash which only has an overhead of 2 to 5 bits per entry.[7]
- When converting the vocabulary from a hash set to a dynamic array (step 2) make sure that the elements that were already inserted to the dynamic array are deleted from the hash set and that the hash set frees some of the memory which becomes unused. This is is a non-trivial process, since decreasing the memory usage of a hash map involves rehashing.
- A similar approach (deleting already moved elements) could be applied when converting the (sorted) array to the hash map.

However, those methods can not deal with the problem that the whole vocabulary is stored in RAM at once. Since our goal was to make it possible to build a QLever index on any machine that is also able to use this index this is not sufficient because of the possible externalization of memory-consuming tokens that still would be applied after the mapping of the vocabulary to the id space was finished. We decided to use a more involved method to meet this requirement which will be presented in the following section.

5.4.2 Improved Implementation

To reduce the memory usage during the creation of the vocabulary mapping we proposed and implemented the following algorithm:

1. Define an integer constant LINES_PER_PARTIAL.
2. Split the knowledge base into disjoint parts $p_1, \dots p_k$ with LINES_PER_PARTIAL triples each.
3. Use the algorithm described in section 5.4.1 to create partial sorted vocabularies $v_1, \dots v_k$ such that v_i contains all the tokens occuring in p_i.
4. Perform a k-way merge on the partial sorted vocabularies to create the complete sorted vocabulary v. From this vocabulary we can directly determine the complete vocabulary mapping m by assigning each token its position in the vocabulary.
5. For each partial vocabulary v_i create a partial vocabulary mapping m_i such that each word is mapped to its id according to the complete mapping m.

[6] `https://github.com/sparsehash/sparsehash`
[7] ibidem

6. For each knowledge base part p_i use the partial mapping m_i to map them to index space.

When considering the RAM usage of this approach we observe the following: During step 2 the size of the data structures (hash set + dynamic array, see section 5.4.1) needed to create the partial vocabulary v_i is upper bounded by the number of distinct tokens in p_i which is upper bounded by $3 \times$ LINES_PER_PARTIAL (each triple introduces at most three new tokens). After v_i is created it is written to disk. This means we only have to handle one partial vocabulary at a time and control the memory usage of this step by the choice of the constant LINES_PER_PARTIAL. Same goes for step 6.
The k-way merge and the creation of the partial mappings (step 3 and 4) are performed in one go as follows:

Algorithm 1: Merging the Partial Vocabularies to Obtain the Complete Vocabulary and the Partial Mappings.

Data: $v_1, \ldots, v_n$, partial vocabularies
Result: $m_1, \ldots m_n$, mappings from tokens to ids and v_{global}, the global vocabulary

```
1  Let Q ← an empty PriorityQueue;
2  Let curId ← 0;
3  Let prevWord ← null;
4  Let m_1, ... m_n be empty mappings;
5  for Each i in {1, ... n} do
6      // Insert first word of each partial vocab together with id of vocab
7      Insert (v_i.first, i) into Q;
8  end
9  while Q is not empty do
10     Let (curWord, originId) ← Q.getMin();
11     Q.deleteMin() ;
12     // We might see the same word in different partial vocabs
13     if curWord ≠ prevWord then
14         curWord ← prevWord;
15         v_global.append(curWord);
16         m_originId[curWord] = curId;
17         curId ← curId + 1;
18     else
19         // Same word has to get the same id
20         m_originId[curWord] = curId − 1;
21     end
22     if v_originId is not exhausted then
23         // Ensure that we always have one word from each partial vocab in Q
24         Insert (v_originId.next, originId) into Q
25     end
26 end
27 return m_1, ..., m_n, v_global
```

Note that in Algorithm 1 we assume the following data structures:

- The PriorityQueue Q stores pairs of strings and ids (the next word from a given partial vocabulary and the index of this vocabulary). It determines its minimum by the lexicographical order of the string. Otherwise it is in an ordinary priority queue that supports insert, getMin and deleteMin operations (implemented for example as a binary heap or a fibonacci heap).
- The sorted vocabularies v_i can be iterated using .first and .next operations that yield the words of the vocabulary in lexicographically increasing order. They are "exhausted" if their last word has already been retrieved and added to the PriorityQueue.

An example of the execution of algorithm 1 can be seen in figure 6.

partial vocabulary p_1		partial vocabulary p_2	
word	**id**	**word**	**id**
alpha	?	bravo	?
bravo	?	charlie	?
delta	?	echo	?
echo	?	-	-

partial vocabulary p_1		partial vocabulary p_2		complete vocabulary v
word	**id**	**word**	**id**	**word**
alpha	0			alpha
bravo	1	bravo	1	bravo
		charlie	2	charlie
delta	3			delta
echo	4	echo	4	echo

Figure 6: Two partial vocabularies before and after merging them. Before the merge (top row) the partial vocabularies are sorted within themselves but the global id of their elements is yet unknown. After the merge (bottom row) the complete vocabulary is created and thus the ids for all the tokens are known.

Our actual implementation of this algorithm additionally has the following properties:

- The complete vocabulary v_{global} is directly written to disk during the k-way merge. It is serialized as a text file with one word per line, sorted in lexicographical order.
- The partial vocabularies v_i are read directly from disk and the mappings m_i are written directly to disk.

Summarized, this step of the vocabulary creation almost requires no RAM since the only data structure that is kept in memory is the priority queue Q. It always contains at most one word per partial vocabulary. This number is typically very small (< 200 during our tests with the full Wikidata knowledge base).

The runtime of the k-way merge is mostly determined by the total size of the partial vocabularies because they have to be completely written to and read from disk. It is not possible to theoretically upper bound this number using the size of the complete vocabulary: It is possible to create worst-case examples where almost all words occur in all partial vocabularies. However, in N-Triples or Turtle dumps of real-life knowledge bases most tokens show a high locality. Most

literals only occur in few triples and the statements about a certain entity are normally close to each other. Then the size of the complete vocabulary is roughly the size of the concatenated partial vocabularies.

5.5 Evaluation and Summary

We have shown how the RAM usage of the vocabulary in the QLever engine can be reduced by externalizing rarely used tokens to disk. This technique reduces the in-memory size of Wikidata's vocabulary from initially 80GB to 20GB. We have also introduced an algorithm which is able to create the vocabulary representation for huge knowledge bases in a memory efficient way. Especially the amount of RAM needed to build the vocabulary is now less than the amount of RAM needed to use it in the QLever engine. This fulfills our goal to be able to build a QLever index on the same machine where we will use it.

6 Compression of Common Prefixes

When looking at large knowledge bases like Wikidata we observe that many of the URIs used as entity names have a common beginning. For example all entity names that were introduced by the Wikidata project start with `<http://www.wikidata.org/`. In the Turtle format for storing RDF triples and in the SPARQL query language we can shorten these common prefixes by using a prefix directive like

```
PREFIX wd: <http://www.wikidata.org/entity/>
```

so that in the rest of the knowledge base or query we can write e.g. `wd:Q5` instead of `<http://www.wikidata.org/entity/Q5>` (see section 2.2). Inspired by this method we came up with the following approach to compress our vocabulary using code words of constant size for a set of prefixes:

- Let C be a set of strings of fixed length called the *codes*.
- Let P be a set of strings or prefixes with $|P| = |C|$ and $code : P \to C$ a bijective mapping of the prefixes to the codes.
- Let len be the function that maps a string of characters to its length.
- If a word x from our vocabulary starts with $p \in P$, so if $x = p \cdot rem$ where $\cdot$ is the concatenation of strings and rem an arbitrary string, store x as $x_c := code(p) \cdot rem$. This saves us $len(p) - len(code(p))$ bytes.
- If there are multiple pairwise distinct $p_1, \dots p_n \in P$ that all are a prefix of x always choose the longest p_i for compression (since we have a fixed length of the codes this saves us the most bytes).

When we want to retrieve a word from the vocabulary, we have to check if it starts with one of our codes $c \in C$. If this is the case we have to replace this prefix by $code^{-1}(c)$ to obtain our original string. In the following we will refer to this decompression process also as the *expansion* of the

compressed prefixes. In order to make this compression process work correctly we have to make sure of the following properties:

- When expanding the prefixes we have to know whether a string was compressed or not. That means that none of the (uncompressed) strings we want to store may start with one of our prefixes from P. If we for example know that our characters will only come from from the ASCII range $(0 - 127)$ we can use byte values from 128 to 255 to encode the prefixes. For the vocabulary of a RDF knowledge base this property is true since all the tokens either have to start with `"` (literals) or with `<` (fully qualified URIs).[8] To be more explicit we have chosen a different approach that prefixes every string with a special code even if it cannot be compressed. Theoretically this could add to the size of our vocabulary for these strings. But for the reasons just mentioned we can easily add `"` and `<` to our set of prefixes so that we at least don't increase our vocabulary size for words where we find no prefix that compresses them better (Assuming that the code length for these prefixes is 1 which is the case in our implementation).
- When expanding the prefixes we need to be able to uniquely identify which code was used to compress our prefix. This holds for example if we use codes of fixed length or so-called *prefix free codes* where no code is a prefix of each other.

In our implementation we have chosen a fixed code length of 1 byte and have used 127 prefixes.

6.1 Finding Prefixes Suitable for Compression

We have seen how a set of common prefixes can be used to compress the vocabulary of a knowledge base. In this section we will describe methods to find a set of such prefixes. The easiest way is to manually specify them. For example when working with the Wikidata knowledge base it is save to assume that it is helpful to compress using the prefixes that are also typically used when formulating queries on this dataset like

```
<http://www.wikidata.org/entity/
<http://www.wikidata.org/entity/statement/
<http://www.wikidata.org/prop/direct/
```

However this approach is not very convenient since it depends on the structure of the knowledge base and has to be executed manually. Additionally we can never guarantee that the prefixes that were chosen are at least close to optimal for compression. Thus we have developed a greedy algorithm to automatically find such prefixes. We will describe this algorithm in the following sections.

6.2 Problem Definition

We have given a set of strings called the *vocabulary* that contains n_v elements. We have given a fixed code length l_c and an integer n_c that describes the number of available codes. We want to find a set P consisting of n_c prefixes such that compressing our vocabulary using the method

[8] QLever originally works on fully qualified URIs and converts the single quotes ' that are also valid for RDF literals to double quotes " to unify the internal representation of tokens.

described at page 20 reduces its sizes as much as possible. We measure this reduction using the *compression ratio* which is calculated by dividing the size of the vocabulary after the compression by its initial size. E.g. a compression ratio of 0.5 means that we halved the sized of the vocabulary.

6.3 Previous Work

According to Fraenkel et al. ([4]) there exists a polynomial time algorithm for the optimal solution of this problem. Unfortunately they give no detailed description of this algorithm or a more accurate measure of its runtime. However, if we assume that in this case "polynomial time" means $\mathcal{O}(n^2)$ or slower we can argue that this algorithm could never be feasible for our needs. Even if we assume that we have to perform exactly n^2 operations for $n = 420 \cdot 10^6$ (This is the size of the Wikidata vocabulary after externalization has been performed as described in section 5.2.2) and each operation takes 1 nanosecond (which is obviously underestimated for operations working on strings) we still need about 5.6 years of time to finish the algorithm.

The algorithm described here was inspired by a heuristic for the same problem that was developed by T. Radhakrishnan in 1978 ([8]). Radhakrishnan also uses a tree to store prefixes that are candidates for compression. However our approach differ's from Radhakrishnan's in the following properties:

- Radhakrishnan uses the per-word compression ratio as a metric while we consider the absolute gain of the prefixes.
- Radhakrishnan does not take into consideration that the compression gain which a certain prefix can achieve highly depends on which prefixes have already been chosen for compression previously. This is an important part of our algorithm.

6.4 Description of the Algorithm

We have implemented a greedy approximation of the set of optimal compression prefixes. On a high level of abstraction this can be described as follows:

Algorithm 2: *FindPrefixes*: Greedy heuristic for finding compression prefixes. High-level description

Data: Vocabulary v (a set of strings), number n_c of prefixes we want to compute
Result: A set of strings P with $|P| \leq n_c$ that can be used for prefix compression on v

```
Let P ← ∅;
while |P| < n_c do
    Let p_next ← the prefix string that compresses the most bytes in v under the
      assumption that we have already compressed v using the prefixes from P;
    if p_next == "" then
        break;
    else
        Let P ← P ∪ p_next;
    end
end
return P;
```

Please note that the break condition $p_{next} ==$"" is only needed to avoid infinite loops in very small vocabularies where $|v| < n_c$ or in the case that all strings that could still be compressed are shorter that the code length. Both cases typically never occur when compressing large vocabularies.

Our next step will be to formalize the way that p_{next} is computed. We will first illustrate this by an example: Consider a vocabulary V consisting of the words *peal*, *pear*, *bin*. Assume we want to find two prefixes for compression that are encoded by 1 character or byte each. Compression using the prefix "peal" will gain us 3 bytes ($(4-1)\cdot 1$, the prefix has length 4, 1 byte is used for encoding, and the prefix occurs in one word), same goes for "pear" while *bin* has gain of 2 ($(3-1)\cdot 1$). The prefix "pea" gains us 4 bytes ($(3-1)\cdot 2$ since it occurs in two words) and thus will be chosen in the first step, so `P={"pea"}` (We have omitted the gain for some possible prefixes that are obviously less than the ones calculated).

In the next step note that the gain for the prefix "peal" has decreased: Compressing its 4 characters originally gained us $4-1=3$ bytes, but since we have already chosen the prefix "pea" for compression we "lose" the 3 characters we originally gained when compressing with "pea". So in this step, choosing the prefix "peal" would gain us only 1 byte, same goes for "pear". We end up choosing "bin" as the second compression prefix which gains us the full $2=3-1$ bytes since none of its prefixes have yet been chosen for compression (We will use this example throughout this chapter).

This example motivates the following definitions which we already have implicitly used:

Definitions

The following definitions all refer to a set of strings *voc* called the *vocabulary* and a set of strings P called the *compression prefixes*. We additionally use an integer constant l_{code} called the *(fixed) code length*.

Definition 6.1. Let s be an arbitrary string. We define the **relevant length** $l_{relevant}(s, P)$ of s by

$$l_{relevant}(s, P) := length(s) - length(maxPrefix(s, P))$$

Where the function *length* returns the length of a string and $maxPrefix(s, P)$ is the longest string in P that is a prefix of s or the empty string if no such string exists.
The relevant length of s is the number of characters we can save for each word in the vocabulary that starts with s when choosing s as an additional compression prefix.

Definition 6.2. Let s be an arbitrary string. The **relevant count** $c_{relevant}(s, voc, P)$ of s is defined by

$$c_{relevant} = |\{v \in voc : startsWith(v, s) \wedge \nexists p \in P : startsWith(v, p) \wedge len(p) \geq len(s))\}|$$

Where the boolean function $startsWith(v, p)$ evaluates to true iff p is a prefix of v.
The relevant count of s is the number of words in our vocabulary which can benefit from additionally choosing s as a compression prefix. This is not the case for words that already were compressed using a longer prefix $p \in P$ (remember that we are designing a greedy algorithm).

Definition 6.3. Let s be an arbitrary string. The **relevant gain** $g_{relevant}(s, voc, P)$ is defined by

$$g_{relevant}(s) := (l_{relevant}(s, P) - l_{code}) \cdot c_{relevant}(s, voc, P)$$

The relevant gain of s is the total number of characters we save in our greedy algorithm when additionally choosing s as a compression prefix.

Now we can formalize the calculation of p_{next} in FindPrefixes (algorithm 2) as:

Algorithm 3: Formal definition of the calculation of p_{next} in FindPrefixes (algorithm 2)
1 Let $p_{next} \leftarrow$ the string that maximizes $g_{relevant}(p_{next}, voc, P)$

It is trivial to see that only strings that are prefixes of words from our vocabulary can have a relevant gain > 0. To efficiently implement the FindPrefixes algorithm we have to calculate the relevant gain of all the prefixes of words from our vocabulary (In the following we will call those prefixes the *candidates*). To do so we need the relevant length and relevant counts of those strings. Both of them possibly change over time since we add to our set P of compression prefixes. In the following we will define a data structure that allows us to efficiently and incrementally compute the relevant gain for all the candidates in each step to find the maximum and thus our next compression prefix p_{next}.

QLever-Tries

Definition 6.4. We define a **QLever-Trie** t to be a rooted directed tree. Since we are only interested in the nodes of the tree we write $n \in t$ for the nodes of the tree. The connection between the nodes is defined by a function *parent* that maps each node except for the root to its unique parent. Each node $n \in t$ is a tuple $n = \{val, num, len, marked\}$ where *val* is a string called the *value* of n, *num* and *len* are integers ≥ 0 and *marked* is a boolean flag. As a convention we write $val(n)$ etc. for the single tuple elements of a given node n. A QLever-Trie t holds the following invariants (in addition to the "ordinary" invariants of a rooted tree):

1. The value of the root of t is the empty string "".
2. The values of the nodes in t are unique.
3. For each node n that is not the root the value of $parent(n)$ is the value of n without the last character (E.g. if $val(n) =$"peace" then $val(parent(n)) =$"peac"

Because of property 3 we do not have to store the complete string value for a node n but it suffices to store the one character we have to append to its parent's value to obtain $val(n)$. Figure 7 illustrates this concept.

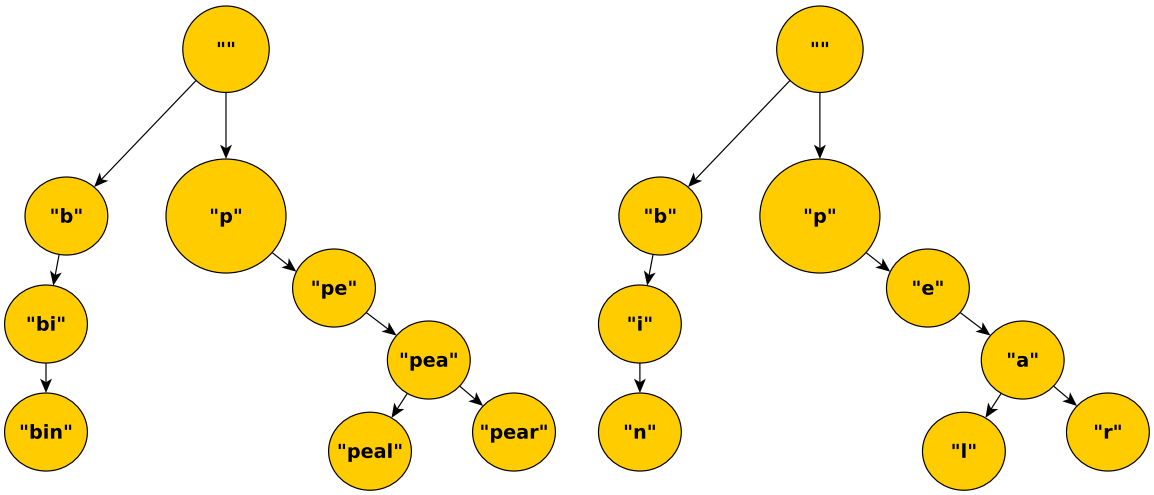

Figure 7: A simple Trie. Only the string values of the nodes are shown. On the left we see the full values that the nodes represent, on the right we see the compressed form with one character per node.

The QLever-Trie is an extension of the general Trie data structure that was originally introduced by de la Briandaise in 1959 ([3]) and is widely used in the field of information retrieval.

To connect a QLever-Trie to our FindPrefixes algorithm we use the following definition:

Definition 6.5. A QLever-Trie t is **associated** with a vocabulary voc and a set of compression prefixes P if all of the following properties hold:

1. $v \in voc \Leftrightarrow \exists n \in t : val(n) = v$
2. $\forall n \in t : num(n) = c_{relevant}(val(n), voc, P)$
3. $\forall n \in t : len(n) = len_{relevant}(val(n), P)$
4. $\forall n \in t : (marked(n) \Leftrightarrow val(n) \in P \vee val(n) =$"")

Properties 2 and 3 ensure that we can calculate the relevant gain of a node n (more exactly: the relevant gain of $val(n)$) by only looking at that node. Property 1 ensures that the trie contains

exactly our set of candidates. Property 4 ensures that all prefixes that have already been chosen for compression and the root are marked. We will see later why this is useful.

To start our algorithm we have to build a QLever-Trie that is associated with our vocabulary *voc* and the initially empty set of compressed prefixes $P = \emptyset$:

Algorithm 4: Building the initial QLever-Trie for the FindPrefixes algorithm

```
Data: Vocabulary voc (a set of strings)
Result: QLever-Trie t that is associated with voc and P = ∅
Let t ← an empty QLever-Trie;
for Each word w in the vocabulary voc do
    for Each prefix p of w (ascending length of p) do
        if p is already contained in t then
            find the corresponding node n and set num(n) ← num(n) + 1;
        else
            insert a new node n into t with val(n) ← p, num(n) ← 1, len(n) ← length(p)
        end
    end
end
Let marked(root(t)) ← true and marked(n) ← false for all other nodes;
return t;
```

By "p is already contained in t" we mean that there exists a node $n \in t$ with $val(t) = p$. It is easy to see that this algorithm produces a trie that is correctly associated with *voc* and $P = \emptyset$:

- Each prefix of each word $w \in voc$ is contained in the trie (trivial) and there are no nodes that do not represent prefixes of words from *voc*.
- $P = \emptyset$ means that we haven't chosen any prefixes for compression yet. So the relevant length of every string is equal to its actual length.
- For each node n, $num(n)$ is the number of words in the vocabulary that start with $val(n)$. Since $P = \emptyset$ this is equal to the relevant count $c_{relevant}(val(n), voc, \emptyset)$.

Once we have created this initial trie, the maximum of the relevant gain and the node/prefix that achieves it (called p_{next} in algorithm 2) can be calculated as follows:

- Traverse the tree.
- Calculate the relevant gain for each node using its relevant length.
- Keep track of the maximum and the node where it is achieved.

An example for a QLever-Trie and the calculation of the maximum can be seen in figure 8.
This gives us the node n_{max} with the maximum relevant gain and thus the prefix $p_{max} = p_{next}$ that we have to add to our set P in the FindPrefixes algorithm. Because of the observations made above it is clear that this step does not only work for the first step ($P = \emptyset$) but always correctly determines p_{next} as long as t is correctly associated with the current set of compressed prefixes P. As a last step we have to make sure that after adding p_{next} to P our Trie t is associated with

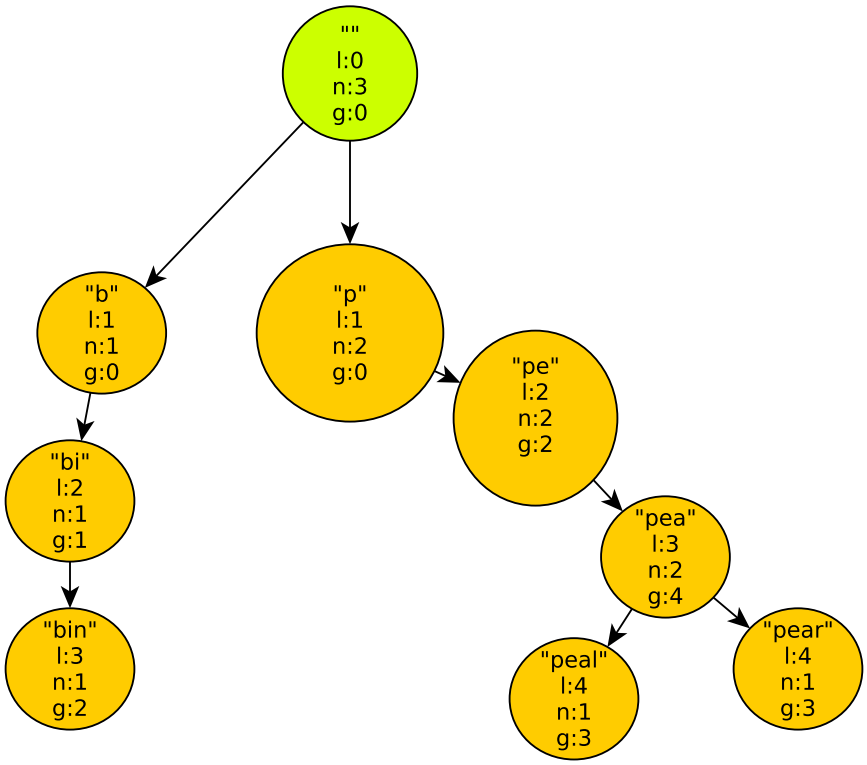

Figure 8: A QLever-Trie for the vocabulary consisting of *peal, pear, bin* for the first step of the FindPrefixes algorithm. Each node stores the prefix string it represents, the number n of occurences of this prefix within the vocabulary, its relevant length l (initially the length of the prefix). The gain g of a node is computed by $(l-1) \cdot n$ (assuming 1 byte of code length). The root is marked (different color)

voc and the updated P again. This is done by algorithm 5 on page 28. This algorithm is called **ReAssociate**.

An example for the application of **ReAssociate** (algorithm 5) can be found in figure 9.

The first part of this algorithm (the reduction of *len* in the subtree of n_{max}) ensures that the *len* of each trie node is equal to the *relevant length* of the prefix string it represents (see definition 6.1 on page 24). Of course we also have to apply this for n_{max}: We set its relevant length to 0 to mark that this node already was chosen. The reason for the **PenaltizeParent** step can also be seen in the example: When we already have compressed using the prefix "pea" and in a later step we additionally compress by "pe" then we obviously do not gain anything for the words that also start with "pea" (Of course this only works with a fixed code length which we always assume for our algorithm FindPrefixes).

6.5 Improvements of the Algorithm

In this section we will describe some improvements for the FindPrefixes algorithm which help reducing its runtime and memory usage.

Algorithm 5: ReAssociate: Modification of the QLever-Trie After Choosing a Prefix

Data: QLever-Trie t, node $n_{max} \in t$ whose value $p_{max} = val(n_{max})$ we have chosen for compression in the current step of FindPrefixes. t

Result: None, modifies t in place. If t was associated with a set $P \setminus \{p_{max}\}$ before the execution of this algorithm, then it is associated with $P \cup \{p_{max}\}$ afterwards.

```
for each node n_desc in the subtree of n_max do
    // adjust the relevant length of each child
    Let len(n_desc) ← min(len(n_desc, length(value(n_desc) − length(p_max))));
end
PenaltizeParent (n_max, lenght(p_max));
marked(n_max) ← true;
```

Algorithm 6: The Recursive *PenaltizeParent* Helper Function

```
function PenaltizeParent (trie-node n, integer penalty)
if marked(n) then
    return
else
    num(n) ← num(n) − penalty ;
    PenaltizeParent(parent(n))
end
```

Only Inserting Relevant Prefixes

In the algorithm just presented all possible prefixes of words in our vocabulary are stored within our tree (e.g. for the word "plum" we have to add "p", "pl", "plu", "plum"). But we observe that not all of them will ever become relevant for our algorithm: If every word that starts with a prefix p_1 also starts with a longer prefix p_2 then our algorithm will never chosse p_1 for compression (see the example vocabulary *peal*, *pear*, *bing* from figure 8 on page 27 where every word that starts with *"pe"* also starts with *"pea"*). It is possible to avoid inserting those superfluous prefixes. To do so we have to adjust the definition of our QLever-Trie (see page 25) in the following way: We replace the condition

- For each node n that is not the root the value of $parent(n)$ is the value of n without the last character (E.g. if $val(n) =$"peace" then $val(parent(n)) =$"peac"

by

- $\forall n \in t : val(parent(n))$ is a prefix of $val(n)$
- If $n_1, n_2 \in t$ and p_{common} is the longest common prefix between $val(n_1)$ and $val(n_2)$ then there is also $n_{common} \in t$ with $val(n_{common}) = p_{common}$.

We will refer to this data structure (the QLever-Trie with the modified conditions) as a **QLever-Patricia-Trie**. It is a modification of the PATRICIA-Trie introduced by D.R.Morrison in 1968 ([6]). We can build a QLever-Patricia-Trie for the prefixes of a vocabulary *voc* as seen in algorithm 7

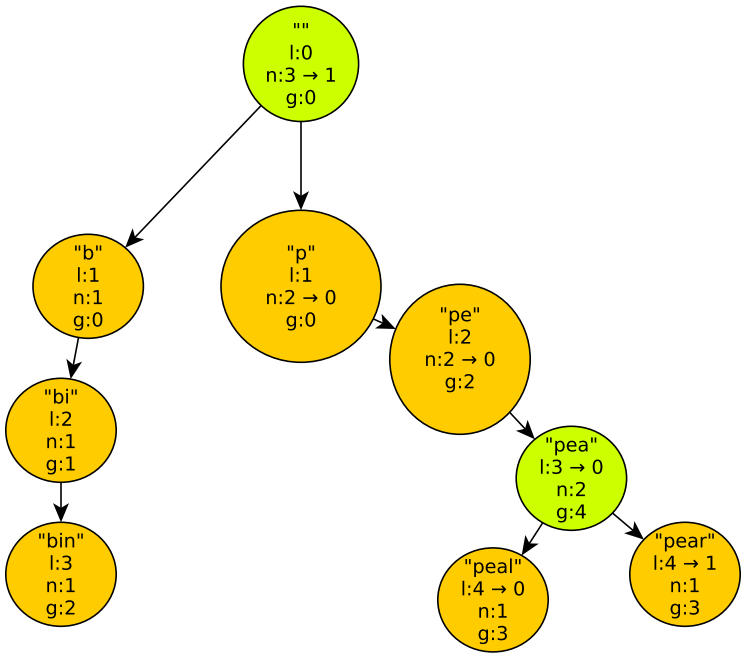

Figure 9: Update of the Trie from figure 8 after the Prefix "pea" has been chosen for compression. $n : x \rightarrow y$ means that the value n has been modified from x to the new value y

Algorithm 7: Building the initial QLever-Patricia-Trie for the FindPrefixes algorithm

Data: Vocabulary *voc* (a set of strings)
Result: QLever-Patricia-Trie t that is associated with *voc* and $P = \emptyset$

Let $t \leftarrow$ an empty QLever-Patricia-Trie;
for *Each word w in the vocabulary voc* **do**
  if *w is already contained in t* **then**
    find the corresponding node n and set $num(n) \leftarrow num(n) + 1$;
  else
    if $\exists n_{conflict} \in t : \nexists n_2 \in t : val(n_2) = commonPrefix(w, val(n_{conflict}))$ **then**
      Let $p_{common} \leftarrow commonPrefix(w, val(n_{conflict}))$;
      insert n_{common} with $val(n_{common}) \leftarrow p_{common}$ and $num(n_{common}) \leftarrow 0$ into t ;
    end
    insert a new node n_{new} into t with $val(n_{new}) \leftarrow w$,
     $num(n_{new}) \leftarrow 1, len(n_{new}) \leftarrow len(w)$
  end
end
Let $marked(root(t)) \leftarrow true$ and $marked(n) = false$ for all other nodes n;
return t;

The function $commonPrefix$ maps a set of strings to the longest string that is a prefix of all the members of the set.

When we have to insert the n_{common} node in our algorithm we observe that $n_{conflict}$ and n_{new} become direct children of n_{common}. An example for this procedure can be seen in figure 10.

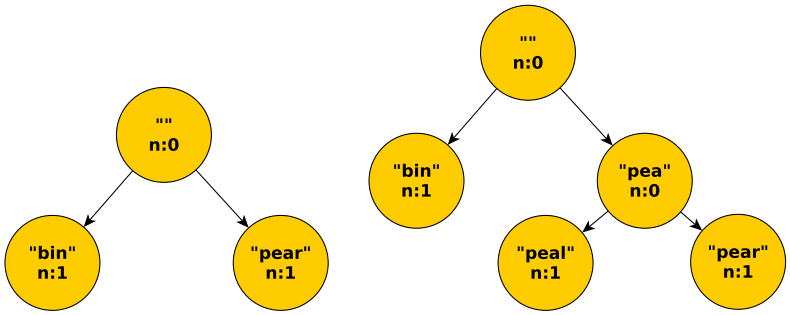

Figure 10: Building a Patricia Tree for the vocabulary {"bin", "peal", "pear"}
. When we have inserted "bin" and "pear" (left half) and we want to insert "pear" we also have to add a dummy node with the value "pea" (right half). Summing up the n-values in the subtree of a node gives us the number of words that start with this prefix. E.g. there are 2 words that start with "pea"

Our Patricia-QLever-Trie t now has the following properties:

- A string s is the longest common prefix of a set of words from our vocabulary if and only if it is contained in t.
- For each node n with value s in the Trie the number of words in our vocabulary of which s is a prefix is the sum of the nums in the subtree of n (including n itself).

Strictly speaking the tree created by algorithm 7 is not associated with voc and $\emptyset$ according to the definition on page 25. But the observation we just made allows us to extend this definition to Patricia-QLever-Tries:

Definition 6.6. A Patricia-QLever-Trie t is **associated** with a vocabulary voc and a set of compression prefixes P if all of the following properties hold:

- $v \subset voc \Leftrightarrow \exists n \in t : val(n) = commonPrefix(v)$
- $\forall n \in t : c_{relevant}(val(n), voc, P) = \sum_{n \in desc(n)} num(n)$ where $desc(n)$ is the set of all descendants of n (including n itself).
- $\forall n \in t : len(n) = len_{relevant}(val(n), P)$
- $\forall n \in t : (marked(n) \Leftrightarrow val(n) \in P \vee val(n) = \text{“”})$

We observe that the ReAssociate step (algorithm 5) also works correctly on Patricia-QLever-Tries. When calculating the relevant gain of a node we have to first sum the counts (num) of all the nodes in its subtree. This can be done in linear time (linear in the number of nodes in the trie) by a postorder traversal.

Only Insert Shared Prefixes

Until now we always have inserted the complete words from our vocabulary into the trie. Typically strings that are only a prefix of exactly one word in our vocabulary are not relevant for our prefix compression since we deal with large vocabularies. Thus in the implementation of this algorithm for QLever we only add prefixes into the trie that are shared between at least two words. Since our vocabulary is alphabetically sorted this can be done efficiently since the following property holds:

- Let w be a word from the vocabulary and $w-$ the word that stands directly before w and $w+$ the word that stands directly after w in our vocabulary.
- If p is a prefix of w but not of $w-$ or of $w+$ then p also is not a prefix of any other word from our vocabulary.

Thus we can reduce the memory footprint of our algorithm by modifying the first step of our algorithm as follows:

- Start with an empty trie.
- For each word w in the vocabulary calculate the longest common prefix $p-$ of w and $w-$ and the longest common prefix $p+$ of w and $w+$.
- Insert the longest string of $\{p+, p-\}$ into the trie.

While this clearly decreases the compression gain of small vocabularies with great length differences (as an extreme example consider `{"a", "b", "thisIsAVeryLongWord"}` as a vocabulary) we have found this improvement to be really helpful with reducing the memory usage and runtime when compressing the vocabulary of huge knowledge bases like Wikidata.

6.6 Evaluation and Summary

In this chapter we have introduced a scheme that uses common prefixes to compress QLever's vocabulary. We also have introduced a greedy algorithm called FindPrefixes that finds prefixes that are suitable for this compression. We have implemented this compression method inside the QLever engine including all the RAM-saving improvements.
After the externalization of large parts of Wikidata's vocabulary the size of the in-memory vocabulary was ca. 20 GB (see section 5.5). We have applied the prefix compression scheme introduced in this chapter on the in-memory vocabulary. It was able to compress ca. 45% of the vocabulary's size so that a size of 11GB remained. The execution of the FindPrefixes algorithm took less than 10 minutes in our setup and required 13GB of RAM (For the hardware setup used in our experiments see section 10.2). Since this time is negligible in comparison to the time needed for the complete index build we have made prefix compression the default setting in QLever.

7 Memory-Efficient Permutations

In this section we will describe improvements for the QLever engine that allow us to efficiently perform SPARQL queries which also contain predicate variables using a much smaller amount of

RAM than the original version of QLever. To do so we first have to introduce *permutations*, the most important data structure used inside QLever and the way they were originally implemented.

7.1 Permutations

To efficiently perform SPARQL queries on RDF knowledge bases Qlever internally uses a data structure called **permutation**. A permutation basically is the set of all triples in a knowledge base, mapped to id space (see section 5) and sorted in a specific way. Each permutation is identified by an actual permutation of the set {Subject, Predicate, Object} or shorter {S, P, O}. For example in the *PSO-permutation* the triples are sorted according to their predicate id. Triples with the same predicate are sorted according to their subject id. If the subject and the predicate of two triples are the same, then the object id determines the order. In the following we say that P is the *first order key* of the PSO permutation while S is the *second order key* and O the *third order key*. Consider for example the following set of "id triples", sorted by different permutations:

S	P	O
0	3	5
1	2	4
1	3	5

S	P	O
1	2	4
0	3	5
1	3	5

Figure 11: A knowledge base of three triples, sorted according to the SPO permutation (left) and the PSO permutation (right)

7.2 Supported Operations

Before we talk about the implementation of the permutation data structure and our improvements to it we will first introduce two operations **scan** and **join** that are needed when executing a SPARQL query and can be performed efficiently using the permutations.

7.2.1 The Scan Operation

The permutation data structure described in the previous section allows us to efficiently retrieve certain subsets of the triples that are already sorted in a specific way. This operation is called a **scan** of a permutation. It can be performed in the following variants:

- **Full scan**. Retrieve all the triples from the knowledge base. Nothing has to be done for this operation because the six permutations already store all possible sortings of this set. Full scans are currently not supported by the QLever engine and also typically not needed for big knowledge bases. Because of this we will only consider the other two scan variants in this thesis.
- **Scan with fixed first order key**. Retrieve all triples with a fixed value for one of the triple elements. For example when we need all triples that have `<sub>` as a subject sorted first by the objects and then by the predicates we perform a scan with the fixed subject `<sub>` on the SOP permutation.

- **Scan with fixed first and second order key**. Retrieve all triples with a fixed value for two of the triple elements. For example when we need all triples that have `<sub>` as a subject and `<pred>` as a predicate we could perform a scan with a fixed subject `<sub>` and a fixed predicate `<pred>` on the SPO or the PSO permutation. The result of both scans would be the same. It depends on the implementation of the permutations (see below) which of the two variants is faster.

For a SPARQL query we typically have to perform one scan per triple. For example the Scan with fixed subject `<sub>` and fixed predicate `<pred>` just described directly gives us the result of the query

```
SELECT ?ob WHERE {
  <sub> <pred> ?ob
}
```

As soon as there are multiple triples in a SPARQL query we additionally need the **join** operation to connect those triples.

7.2.2 The Join Operation

Consider the following SPARQL query consisting of two triples:

```
SELECT ?x WHERE {
  ?x <is-a> <astronaut> .
  ?x <gender> <female>
}
```

In the previous section we have seen how we can translate each of the triples into a **scan** operation. After performing them we end up with

- A list of possible values for `?x` that meet the constraints of the first triple (a list of all astronauts).
- A list of possible values for `?x` that meet the constraints of the second triple (a list of all female entities).

To complete the query we have to intersect those two lists to find all `?x` that are contained in both of them. This operation is referred to as a **join**. Since both of our intermediate result lists are already sorted the join can be performed in linear time using the intersection algorithm for sorted lists. With more complex queries sometimes the intermediate results are not automatically sorted in the way needed for the join. In this case the QLever engine first has to sort them. There can be also more complex variants of the **join** operation e.g. when multiple variables per triple are involved. These can also be performed using similar approaches (sort if necessary and intersect).

7.3 Implementation

The implementation of the permutations makes use of the fact that all triples with the same first order key are stored adjacent to each other. Thus it suffices to store only the second and third order key of each triple. For the first order key we only have to store the index of the first triple (the offset) for this key and the number of triples for that key (see figure 12).

S	P	O
1	2	4
0	3	5
1	3	5

S	O
1	4
0	5
1	5

$\}P = 2$, offset $= 0, num_{triples} = 1$

$\}P = 3$, offset $= 1, num_{triples} = 2$

Figure 12: A small knowledge base sorted according to the PSO permutation. On the left we see the complete triples. On the right we see QLever's implementation: For the contiguous range of each first order key (P in this example) QLever only stores the range of the triples.

7.3.1 Original Implementation

The original version of QLever stores the second and third order keys of each triple (S and O columns in figure 12) in a contiguous area of disk memory. The offset and the number of triples (this information is called a `FullRelationMetaData` in the source code) for each first order key are stored in a hash table that maps the id of the first order key to its `FullRelationMetaData`. Using this implementation of the permutations we can perform the different variants of the scan operation (see section 7.2.1) as follows:

- Scan with fixed first order key k: First get the offset o and the number of triples n for k from the hash table ($\mathcal{O}(1)$ in average) and then read all the triples in the range $[o, o + n)$ from hard disk ($\mathcal{O}(n)$ with the overhead of the hard disk access).
- Scan with fixed first and second order key k_1, k_2: First get o_1 and n_1 for k_1 as stated above. Within the range $[o_1, o_1 + n_1]$ on hard disk find the range $[o_{1,2}, n_{1,2}]$ that additionally has the correct k_2 using binary search. We then read this range ($\mathcal{O}(n_{1,2} + \log n_1)$ with the overhead of the disk access).

For big relations binary search on disk is not efficient since a lot of different memory pages have to be touched. Additionally magnetic hard drives are very slow when it comes to random access. To deal with this issue the original version of QLever also implements *blocks* which are basically skip pointers into big relations that are stored in RAM. The details of this implementation can be found in [1], page 3.

7.3.2 Issues of the Original Implementation

We have found that the access to the second and third order keys on hard disk using the blocks/ skip pointers works very efficiently also for large-scale knowledge bases. However we ran into trouble when building the hash map for several permutations of the full Wikidata knowledge base: Wikidata contains about 1 billion different objects and subjects each, so we needed to store 1 billion entries in the hash table for each of the OSP, OPS, SPO and SOP permutations. For each entry we had to store at least 32 bytes (size of a single id + size of a single `FullRelationMetaData`) which already gives us more than 30GB per permutation if we assume a hash table without any memory overhead which is clearly utopian. Since we need four hash tables of this size for the different permutations it is easy to imagine that even machines with 200GB of RAM would struggle with an index of the full Wikidata knowledge base (including the overhead of hash tables and other information like the vocabulary which we additionally store in RAM, see section 5). This implementation worked well for the original purpose of QLever: It was designed to work on

the special SPARQL+Text extension. There it is very uncommon to have predicate variables in SPARQL queries. When having a fixed value as predicate in every triple of a query it suffices to only build and work with the POS and PSO permutations. For those permutations the hash table approach just described works well since even large knowledge bases typically only have a relatively small number of predicates (ca. 30000 for Wikidata, ca. 700000 for Freebase) which can be stored in a hash table without running into trouble.

7.3.3 Improved Implementation

In the previous section we have seen that storing the `FullRelationMetaData` of certain permutations in RAM is not feasible for large knowledge bases. This directly suggests externalizing them to hard disk and only loading them as needed. Theoretically we could implement or use a disk-based hash table but this is not necessary for the following reasons:

- Hash tables are efficient and necessary when the set of actually used keys is much smaller than the set of possible keys (e.g. when using arbitrary strings as keys for a hash table the set of possible keys is infinite).
- For our purpose the keys are integers which are always less than the total vocabulary size of the knowledge base (see section 5).
- For the critical permutations OSP, OPS, SPO and SOP the number of first order keys that have to be stored in the hash map is almost equal to the size of the complete vocabulary.

These observations suggest the following implementation for a dictionary where the set of possible keys is a range of integers (called $possibleKeys$). We will refer to this data structure as a *dense dictionary*.

- We need a special value called the *empty value* that will never occur in the data we store in the dictionary.
- We create an array arr with size $|possibleKeys|$ and fill it with the empty value.
- When inserting a $(key, value)$ pair we write $value$ to the array position $arr[key]$.
- When checking for a key in the dictionary we access $arr[key]$ and check if the found value is not the empty value.

All other operations typically supported by a dictionary can be implemented in a similar way. This data structure which is basically just an array with special operations can easily be implemented on disk. For this purpose we have first implemented a templated data structure `MmapVector` that is a dynamic array which stores its information in a persistent file on hard disk. This file is mapped to a virtual memory address space using the POSIX system call `mmap`. That way the actual reading of the file and all caching and buffering mechanisms are handled by the operating system (OS). A nice side effect of using `mmap` is that the OS can automatically cache and pre-load a huge portion of the file if there is a lot of RAM available. If this is not the case it suffices to only load pages of the file that are actually accessed and immediatly remove them from memory as soon as the access is completed. That way we can create huge dynamic arrays on systems with

a small amount of RAM available but on the other hand our application possibly gets faster when there is more RAM available since the number of page misses should be less.
Based on the `MmapVector` we have written a dictionary-like type called the `MetaDataHandlerMmap` that implements a *dense dictionary* for `FullRelationMetaData`. We then use this type as a template parameter for our permutations. That way we can keep the hash table implementation for the permutations where it is efficient (PSO and POS) while all other permutations are implemented using our dense and disk-based `MetaDataHandlerMmap`.

7.4 Evaluation and Summary

We have presented a memory-efficient implementation for dense permutations inside QLever (permutations where almost all possible ids appear as a first order key). It replaces a $\mathcal{O}(1)$ access to a hash table in RAM by a $\mathcal{O}(1)$ access on hard disk. We have not found any significant influence on the scan times since the runtime of a scan is mostly determined by reading the actual triples from disk.

8 Efficient Language Filters

Language filters are a feature commonly used in SPARQL queries on Wikidata and other knowledge bases that contain literals in different languages. In this chapter we will first give an introduction to the syntax and semantics of language filters in the SPARQL language. Then we will discuss several possible implementations of this feature for the QLever engine.

8.1 Introduction to Language Filters

Many large-scale knowledge bases contain literals in many different languages. For example the following query for the names or labels of the entity Q42 (English writer Douglas Adams) yields more than 100 results on Wikidata:

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label WHERE {
  wd:Q42 rdfs:label ?label
}
```

We observe that the result contains the name of Douglas Adams in many different languages. Many of those literals are equal if we disregard their language tag, e.g. `"Douglas Adams"@en` and `"Douglas Adams"@fr` but the result also contains transliterations of this name in Arabic, Cyrillic, Korean and other non-latin scripts. While the result size is not a problem for this simple query we can observe that the multilinguality of Wikidata easily causes trouble when we have a query that already would yield many results without the `rdfs:label` relation which then bloats up the result size although we are actually interested in only one name for each entity, preferably in a language we understand. This motivates the use of language filters in SPARQL. For example we can formulate the query "What is the english name of the entity *wd:Q42*" in SPARQL as

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label WHERE {
  wd:Q42 rdfs:label ?label .
  FILTER langMatches(lang(?label), "en")
}
```

The function `langMatches` returns true if and only if its first argument (a single language) matches its second argument (a language range, see below). The function `lang(?label)` returns the language of the variable `?label` as specified by its language tag. Note that the language range `"en"` also matches "sub-languages" of English like `"en-gb"` (British English) or `"en-ca"` (Canadian English). If we only want to have exact matches of the language tag we can also write

```
FILTER (lang(?label) = "en")
```

8.2 Possible Implementations

In chapter 5 we have described that the ids of tokens from the knowledge base also reflect their alphabetical or numerical order. This way we can efficiently implement *value filters* like `FILTER ?x < 3` efficiently by checking if the id of `?x` is smaller than the id of the first numerical token that is ≥ 3. (By checking for $\geq$ we also get correct results if the token 3 is not contained in our vocabulary). We can easily see that this filter can not be extended to work for the language tags of the literals since the literals are sorted by their alphabetical order and not by their language. In this section we will describe methods to extend the QLever engine by an efficient mechanism for SPARQL language filters.

8.2.1 Checking the String Value

The probably easiest way to perform a language filter goes as follows:

1. Compute the result of the query (in the id space) without the language filter.
2. Convert all ids to their corresponding string values.
3. For each row in the result, check if the language tag of the variable that is to be filtered matches the desired language tag.

This method was the first to be implemented in QLever since it requires no changes in the structure of the index or the query execution and is (obviously) able to correctly perform the language filtering. However we can already assume that it will not be very efficient because we always have to compute the full result containing all the languages. Additionally we have to translate all the values of the filtered variable to their string value which is a relatively expensive procedure.

8.2.2 Sorting the Vocabulary by Language

This variant is inspired by the implementation of the value filters: If the literals for one language form a consecutive range in the vocabulary and thus in the id space we can directly filter the

language on the level of the ids. To implement this method it would suffice to transform all the literals from the form `"value"@langtag` to `@langtag"value"`. This allows us to determine the language of a literal directly by checking if its id lies in the range of a certain language. This method still has the disadvantage of first having to compute the full and unfiltered result in the id space. Additionally we lose the lexicographical sorting of the literals which means that for example an `ORDER BY`-clause for a result variable with string values would not work properly anymore as soon as our result contains literals in different languages. Of course it would be possible to change the implementation of `ORDER BY` to a probably less efficient method. But it is highly questionable if this would be overall desirable since sorting results of SPARQL queries alphabetically by an `ORDER BY` clause also is a frequently used feature which we want to keep as efficient as possible.

8.2.3 Using a Special Predicate for the Language Tag

This and the following method internally transform the language filter clause into a special RDF triple which is then treated in the "ordinary" way by QLever's query planner.[9] This first approach uses a special predicate called `ql:langtag` and special entity names like `ql:@en` to map each literal to its language. This means that during the index building process when we see a triple containing a literal with a language tag like

```
<subject> <predicate> "french literal"@fr .
```

we additionally add the triple

```
"french literal"@fr ql:langtag ql:@fr .
```

When parsing a SPARQL query with a language filter we similarly transform

```
FILTER (lang(?variable) = "de")
```

to the triple

```
?variable ql:langtag ql:@de .
```

Please note that the special triples we add are actually forbidden by the RDF standard since literals can never occur as subjects of a triple. We have relaxed QLever to internally allow them so that we can evaluate this method of language filtering.
This method also correctly implements language filters. It can be easily integrated into the QLever engine since we only have to modify the query parsing and the triple parsing during the index build. Since the filters are transformed to "ordinary" triples they can also be optimized by QLever's query planner. One disadvantage is the increased size of the index: For every literal that has a language tag we add a triple which increases the size of our permutations. But since those are only stored on disk this does not hurt much. We also do not expect this method to slow down queries which do not use a language filter: In the PSO and POS permutations the newly inserted predicate forms a contiguous range on disk that is only touched when using a language filter. With the other permutations the new triples are added "in the middle" (see section 7.1 for the structure of permutations) which could in theory slow some queries down. But since those permutations typically have only few triples per first order key[10] we do not expect this to have a great impact on our query speed.

[9] For the internals of the query planning algorithm in QLever see [1]
[10] E.G. For a given subject there typically is only a relatively small number of triples with this subject.

8.2.4 Using Language-Tagged Predicates

This approach resembles the one just described as it is also based on adding special triples to the knowledge base. This time we add the language tag to the predicate of the triple a literal occurs in. This means that during the index build when parsing the triple

```
<entity> <label> "english-label"@en .
```

we also add the triple

```
<entity> @en@<label> "english-label"@en .
```

When parsing a language filter on a variable `?x` for the language `"lang"` in a sparql query (`FILTER (lang(?x) = "lang")`) we have to do the following:

1. Find a triple where `?x` occurs as the object and the predicate is *not* a variable. Let `<sub> <pred> ?x .` be that triple (`<sub>` may also be a variable).
2. Substitute `<sub> <pred> ?x` by `<sub> @lang@<pred> ?x .`

We see that this approach can only be applied if the query contains a triple that meets the requirements from step 1. For example it can not execute the following query correctly, be cause the filtered variable only occurs in connection with predicate variables:

```
SELECT ?literal WHERE {
  wd:Q42 ?pred ?literal .
  FILTER (lang(?literal) = "de")
}
```

("Get all German literals that are directly connected to Douglas Adams"). Another type of query that cannot be performed is when the second argument of the `langMatches` function is not a constant:

```
SELECT ?name ?description WHERE {
  wd:Q42 rdfs:label ?name .
  wd:Q42 schema:description ?description .
  FILTER (lang(?name) = lang(?description))
}
```

("Get all pairs of (name, description) for Douglas Adams such that the name and the description have the same language"). But there is also a great benefit of this method: For each predicate p that has string literals as objects in at least one of its triples and for every language tag l we precompute the subset of triples for p filtered by l. Since those triples are stored contiguously on hard disk we never have to deal with the complete relation but can always directly work with the filtered relation as soon as this filtering approach can be applied. As an extreme example for this consider the query

```
SELECT ?x ?label WHERE {
  ?x rdfs:label ?label .
  FILTER (lang(?label) = "en")
}
```

The filter using the `ql:langtag` predicate would translate this to

```
SELECT ?x ?label WHERE {
  ?x rdfs:label ?label .
  ?label ql:lang ql:@en
}
```

The query planner has to perform a join between the full `rdfs:label` relation (from the POS or the PSO permutation) with the result of a scan of the POS permutation with the fixed predicate `ql:lang` and the fixed object `ql:@en` (For the description of the join and scan operations see section 7.2).

The method just described would translate this query to

```
SELECT ?x ?label WHERE {
  ?x @en@rdfs:label ?label .
}
```

So it would do a full scan of the PSO or POS relation with `P = @en@rdfs:label` which would directly yield the result. So we would only load information into RAM that is actually relevant for the result of our query. Although this surely was an extreme example we can argue that our query planner in every query that contains a triple `<sub> <fixed-predicate> ?x` where `?x` is language-filtered with language `"lang"` at some place has to touch the relation for `<fixed-predicate>`. With the language-tagged predicates it can instead use the relation `@en<fixed-predicate>` which is smaller than `<fixed-predicate>`. This substitution has no additional overhead and directly performs the language filter.

8.3 Evaluation and Summary

We have described 4 possible implementations for a language filter in QLever. The first two (filtering the language tags at the end by the actual string values of the result and sorting the vocabulary according to the language tags) have serious disadvantages: The first method is very inefficient while the second one broke the functionality of efficient `ORDER BY` clauses. The most promising approaches use the insertion of special triples and can thus be optimized by QLever's query planner. Introducing a special predicate `ql:langtag` that maps each literal to its language tag is a flexible approach that can correctly handle all possible uses of language filter clauses that are allowed by the RDF standard. Introducing special predicates `@lang@<predicate>` that already filter the relations by language during the index build can only be applied when we filter for an explicitly stated language and if the filtered variable appears together in an triple. We have implemented those last two methods for QLever: The `@en@<predicate>` variant is chosen whenever this is possible otherwise we fall back to the `ql:langtag` approach. In chapter 10 of this thesis there is an evaluation that shows that this choice is the most efficient one.

9 Effective Query Construction on Wikidata

Until now we mainly discussed how to efficiently perform given SPARQL queries on a certain knowledge base using the QLever engine. In practice however it is also a hard task to find the

appropriate query that yields the expected results. When we for example want to find out who currently is the CEO of Apple using a knowledge base we have to find answers to the following questions:

- How is the company *Apple* represented in our knowledge base. For example in Wikidata, Apple (the manufacturer of computers etc.) is represented by `wd:Q312`.
- How is the relation between a company and its CEO modeled in the knowledge base. For example in a knowledge base with readable entity names this could be modeled in one of the following ways:

```
# option 1
<Dummy Inc> <CEO> <Mr. Boss> .
# option 2
<Mr. Boss> <CEO> <Dummy Inc> .
# option 3
<Mr. Boss> <works at> <Dummy Inc> .
<Mr. Boss> <position> <CEO> .
```

The most convenient way to handle this problem would be to automatically translate natural language to SPARQL queries using e.g. deep learning. One system that aims to solve this difficult task is Aqqu ([2]). However such systems are currently only able to handle simple questions and fail with more complex requests.
Another promising approach to make query construction easier is the query autocompletion implemented by J. Bürklin and D. Kemen for the QLever engine. This can be tried out in the QLever-UI at `http://qlever.informatik.uni-freiburg.de/` when choosing the *Wikipedia + Freebase Easy* backend. When typing a SPARQL query this UI suggests entities that lead to a non-empty result given the partial query that has already been typed in. However this currently only works on the Freebase Easy knowledge base that contains human-readable entity names. One step still needed to make this method work on Wikidata is to implement an efficient way to find the correct entity name for a given verbal description (e.g. `wd:Q312` for "Apple Inc."). We have implemented an efficient system that solves this task which we will describe in the following sections.

9.1 Problem Definition

When creating a SPARQL query on Wikidata a major challenge is to find out which of the internal entity and property names represent the entities from our query. For instance, when querying for a list of humans, sorted by their dates of birth we conceptionally could try the following query:

```
SELECT ?person ?dateOfBirth WHERE {
  ?person <is-a> <human> .
  ?person <date-of-birth> ?dateOfBirth . }
ORDER BY ASC(?dateOfBirth)
```

To successfully execute this query on Wikidata, we have to know that the concept of being a member of the human species is assigned to the name `wd:Q5`, the relation "is a" or "instance of"

is represented by the property *P31* and the date of birth can be retrieved using `wd:P569`. Then our query becomes

```
SELECT ?person ?dateOfBirth WHERE {
  ?person wdt:P31 wd:Q5 .
  ?person wdt:P569 ?dateOfBirth
}
ORDER BY ASC(?dateOfBirth)
```

In this section we want to introduce a software we have implemented to effectively find the find the proper Wikidata entity names (Q... and P...) for a given textual representation.

9.2 The Entity Finder

To obtain the correct names for Wikidata entities one can e.g. use the search fields on the Wikidata homepage (`https://www.wikidata.org/`). This typically works well when manually creating relatively small queries. Since we aim to create a Sparql UI for Wikidata using QLever which is completely independent from the Wikidata servers we have implemented an alternative software called the *Entity Finder* that can also efficiently perform this task. It consists of a frontend where the user can type in a query and see and use the created results and a backend that performs the actual search. The Entity Finder supports the following features:

- **Synonyms**. Most entities or concepts are known under various names, even within the same language. For example the English terms "profession" and "occupation" normally refer to the same concept. For the Wikidata knowledge base it is relatively easy to implement synonyms since Wikidata already contains many synonyms (or aliases as they are internally called) for its entities. For instance, Wikidata contains the information, that the entity `wd:Q60` whose preferred English label is *New York City* is also referred to as *NYC*, *City of New York*, *The Big Apple*, ... Within the Entity Finder a given query matches an entity if it matches its preferred label or one of its aliases.

- **Search-As-You-Type**. We wanted the search to be interactive. This means that the user can see intermediate results of their query before having completed it. There is no explicit start button for the search but the results are always refined as the user changes the text in the search field. Making this work requires two components: In the frontend a new query is sent each time the user changes the contents of the search field. In the backend we have implemented a prefix search. This means that a given query matches all names and aliases of which it is a prefix. For example when typing "app" into the search field, "apple" is one of the results.

- **Useful ranking of the results**. Many entities are known under the same name. For example the term "apple" can refer to the fruit or to the IT company. Especially when we enable prefix search (see above) and also take synonyms into account there are often many entities that would match a query. We have implemented a ranking that effectively ranks "expected" results of a query higher but is still fast to compute to not endanger our goal of interactiveness. The details of this ranking are described below.

- **HTTP-API**. The backend of the Entity Finder is implemented as a HTTP server that answers GET requests containing a search term with the corresponding search results. This ensures that the Entity Finder is independent from any concrete user interface and can be easily reused as a building block for different applications that make use of Wikidata entity names.

9.3 Ranking Matches

When ranking search results there are generally two different approaches:

- Assign a score toeach possible search result and rank the results according to this score.
- Determine how well a possible result matches the query and rank according to this metric. An example for such a metric is the Levenshtein edit distance between the query and a possible result.

Typically a mixture of both approaches is applied because we typically want both: A high relevance of the search results and a good matching between our query and the result.
In the Entity Finder we only consider names as a result of a query if the query is a prefix of the name. Additionally we rank all exact matches before all prefix matches. For example when our query is "a" then the name "a" (representing e.g. the first letter of the latin alphabet) is always ranked before "apple". That way we avoid that a longer name with a higher score prevents its prefixes from being found. E.g. if there are many entries that have "apple" as a name and all have a higher score than "a" and we would only rank according to the scores, then we would not be able to ever find the latter. Additionally this method is computationally cheap because the contiguous range of the prefix matches for a given query as well as the range of the exact matches can be found using binary search (We store all names and aliases of entities in a sorted manner). As a simple metric for the score of a given entity in Wikidata we have chosen its number of *sitelinks*. This is the number of websites in all the Wikimedia projects that this entity is linked to. Then our complete ranking algorithm can be summarized as follows:

- Sort all exact matches before all prefix matches.
- Within the exact matches and within the prefix matches sort according to the sitelinks score.

To preserve the fast computation of the ranking for big results we disable the sorting of the prefix matches when our result size exceeds a certain threshold. In this case the query is typically short and unspecific yet so in practice this does not affect the quality of our search results.

9.4 Summary

We have presented the Entity Finder, a fast search engine that interactively finds Wikidata entities by their names and aliases. We have introduced a simple but effective ranking heuristic that we have found to work well for the entities of typical queries and additionally is fast to compute. As a next step it would be possible to combine the Entity Finder with the autocompletion feature from the QLever-UI to also make the autocompletion work with the Wikidata knowledge base. A demo of the Entity Finder can be found at `http://qlever.informatik.uni-freiburg.de/drag-and-drop-ui/`

10 Evaluation

In this section we will evaluate the performance of QLever on the full Wikidata knowledge base against the Wikidata Query Service (WQS). A special focus will be put on the different language filter implementations. We are well aware that this comparison does not replace a proper evaluation of different SPARQL engines that are run in a comparable and deterministic setting. However we still found that the evaluation described here provides useful insights to the performance of QLever and suggests further improvements.

10.1 Benchmark

We have evaluated QLever and the WQS on three queries, those are

- **Q1**: A query searching for all English names of entities that are directly connected to Douglas Adams:

```
SELECT DISTINCT ?y ?yLabel WHERE {
  wd:Q42 ?x ?y .
  ?y rdfs:label ?yLabel .
  FILTER (lang(?yLabel) = "en")
}
```

- **Q2**: A query for German cities and their German names

```
SELECT ?cityLabel ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
  FILTER (lang(?cityLabel) = "de")
}
ORDER BY DESC(?population)
```

- **Q3**: A query for persons and their birth date if the birth date is precise by the day (precision value `"9"`)

```
SELECT ?person ?personLabel ?date_of_birth
WHERE {
  ?person p:P569 ?date_of_birth_statement .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth .
  FILTER (lang(?personLabel) = "en")
}
```

Each query was executed in the following variants:

- With the language filter (as just described)

- Without the language filter (literals in all languages)
- Without retrieving the labels (completely omitting the triple with `rdfs:label`)

For QLever we have additionally evaluated two different language filters: The one where a special predicate `ql:langtag` is used to map each literal to its language and the one where the predicates get annotated versions for each language like `@en@rdfs:label` (see section 8.2 for details).

10.2 Setup

The QLever instance was run on a machine with the following technical specifications:

- CPU: 2x Xeon E5640 2,6 GHZ 4C 12MB
- RAM: 96GB, 12*8 GB DDR3 PC3-10600R ECC
- Disk: 5x 1TB SSD (Samsung Evo 960) as a hardware raid 5

The Wikimedia team reports that the Wikidata Query Service is run on six servers with specifications similar to the following:

- CPU: dual Intel(R) Xeon(R) CPU E5-2620 v3
- RAM: 128GB
- Disk: 800GB raw raided space SSD RAM: 128GB

(see `https://www.mediawiki.org/wiki/Wikidata_Query_Service/Implementation#Hardware`). Each measurement was repeated six times, we report the average values and the standard deviation. In QLever the cache was emptied after each query. In the WQS we waited for 30 minutes after each measurement to limit the effect of caching. For the Wikidata Query Service we did not use actual language filters but the Wikibase Label Service which is the preferred method to retrieve labels of entities. The exact queries that were executed for each measurement are reported in attachement A.

10.3 Results

Figures 13 throuh 15 show the query times of our three benchmark queries Q1, Q2 and Q3 in the different variants. The labels of the data points have the following meaning:

- **QL:@en@** : QLever, language filter implemented by language-tagged predicates (`@en@rdfs:label` etc.)
- **QL:langtag** : QLever, language filter implemented by the special predicate that maps literals to languages (`ql:langtag`)
- **QL:no filter** : QLever, no language filter (literals in all languages)
- **QL:no label** : QLever, no label resolution (only abstract entity names as result)
- **WD** : Wikidata Query Service, with language filter

- **WD:no filter** : Wikidata Query Service, no language filter (literals in all languages)
- **WD:no label** : Wikidata Query Service, no label resolution (only abstract entity names as result)

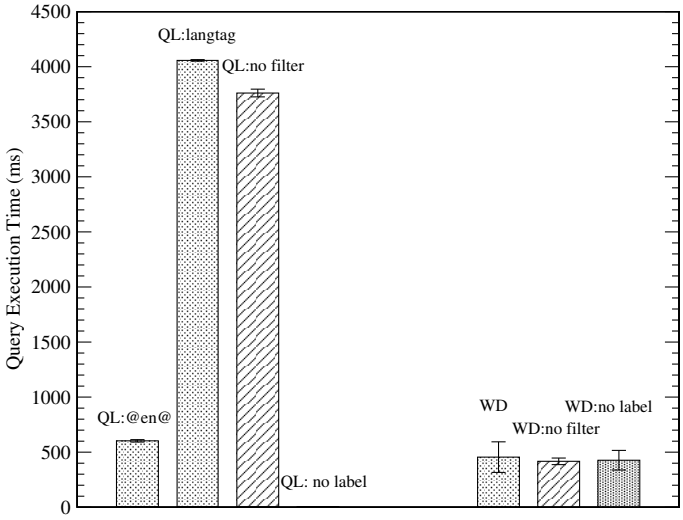

Figure 13: Query Times for query **Q1**, The value of *QL:no label* is 2 ms

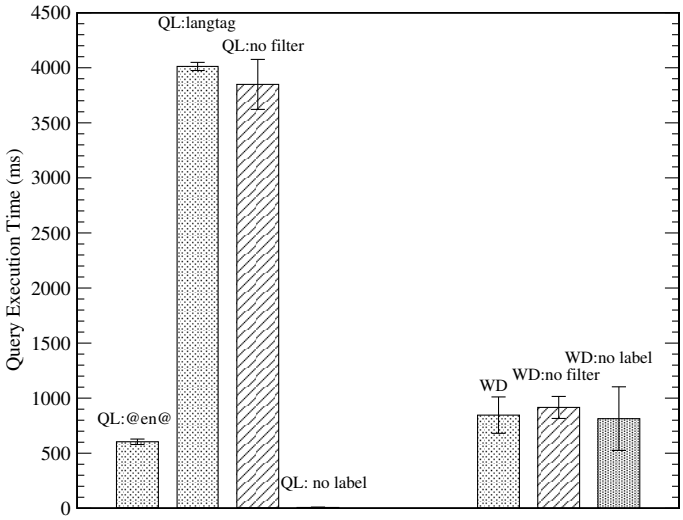

Figure 14: Query Times for query **Q2**, The value of *QL:no label* is 8 ms

Looking at these graphs we can make the following observations: The different language filters in QLever perform as expected. It is always more expensive to retrieve the labels for our query result because it requires an extra join with `rdfs:label`. We get the least overhead with the `@en@rdfs:label` variant of the language filter because it joins with already prefiltered predicates that contain less triples. The language filter using the special `ql:langtag` predicate is even slower than calculating the unfiltered result since it requires two joins (one with `rdfs:label` and one with `ql:langtag`. We also observe that in our example queries QLever spends most of the time

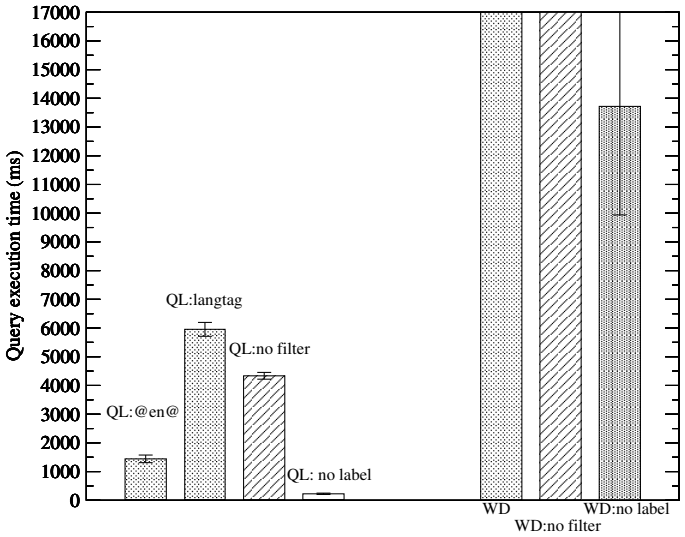

Figure 15: Query Times for query **Q3**, the times for *WD* and *WD:no filter* could not be measured because the Wikidata Query Service always returned a timeout

retrieving the labels. This is not surprising for Q1 and Q2 which are otherwise computationally relatively simple but it also holds for Q3. We can see that this is a complex query by looking at the Wikidata Query Service which struggles or even fails to execute it (see figure 15). When looking at the results of the WQS we see no big differences between the different variants. This suggests that they use a different mechanism for the label retrieval than for the ordinary query processing although we cannot be sure about this without checking the actual implementation. When comparing QLever to the Wikidata Query Service we see that QLever is always two orders of magnitude faster when we do not retrieve labels. Even if those numbers are not really comparable it still speaks for QLever that it is able to efficiently compute the result of Q3 which is beyond the capabilities of the WQS as soon as there are labels involved.

11 Conclusion

In this thesis we have shown that we can use externalization of uncommon tokens and automatic prefix compression to reduce the RAM usage of QLever's vocabulary for the Wikidata dataset. We have also shown that externalizing the index permutations can further reduce the memory footprint of the QLever engine without affecting its speed. Combining those two measures allows QLever to run using 24GB of RAM while it initially failed to build an run a Wikidata index even on machines with $>$ 200GB RAM. We also presented a way to build the vocabulary without ever loading it completely to RAM. That way the memory needed for building the index is lower than the memory needed for running it. Thus we have achieved the most important goals for this thesis. We have also presented an efficient implementation of SPARQL language filters which are heavily used in typical queries on the Wikidata knowledge base. We also discussed which measures have to be taken to allow the effective creation of SPARQL queries for Wikidata using autocompletion.

11.1 Further Work

To further improve the performance and the usability of QLever we can think of the following aspects:

Complete Support of the SPARQL Language

QLever still lacks the support for several features of the SPARQL language like subqueries or certain types of filters. Thus it is currently not possible to execute certain types of queries. Supporting the full SPARQL standard would increase the usability of QLever and also make it easier to systematically evaluate its performance in comparison with other SPARQL engines.

Systematic Performance Evaluation

Since the focus of this work was to reduce the memory footprint of QLever and to make it run on the full Wikidata knowledge base we did not yet find the time to evaluate QLever's performance on this dataset in comparison to other RDF engines. For example it would be interesting to compare QLever's performance on Wikidata to the performance of the Blazegraph engine (`https://www.blazegraph.com`) which is used in the Wikidata Query Service.

Further Compression of the Vocabulary

In this thesis we have only considered the compression of common prefixes. To further reduce the memory footprint of QLever's vocabulary we could implement and evaluate some state-of-the art general-purpose dictionary compression schemes like they are discussed in [7] and [5]. We can think of two motivations for further compression:

- If we can reduce QLever's memory footprint for Wikidata below 16 GB we would allow to locally set up a SPARQL engine for the full Wikidata on a modern consumer PC system (Consumer systems with 32GB of RAM or more are still very uncommon).
- With better compression we could possibly get rid of the externalization of the vocabulary and keep the complete vocabulary in RAM again. This would eliminate the expensive random access to hard disk that currently occurs when a query or a result contain tokens that are externalized.

Faster Mechanism for Label Resolution

We have seen that the language filter implementations for QLever that were presented in this thesis still have a relatively big overhead especially for queries that are otherwise computationally cheap. Since typical queries on Wikidata always retrieve labels to make their results readable to humans if other methods can solve this problem faster. This probably leads to keeping a mapping of all entities to their label in RAM at least for common languages.

Make Query Autocompletion Work on Wikidata

The effective creation of queries on Wikidata or any other large-scale knowledge base with abstract entity representations is hard because `wdt:P1082` is much less readable than `<population>`. We

can think of a well-usable Wikidata-UI that utilizes the following features:

- Display abstract entities together with their name and description. Thus the user is able to perceive the internal structure of Wikidata as well as the human-readable semantics of their query at the same time.
- Use context-aware autocompletion in the query editor that utilizes the human-readable names of entities. Thus when the user starts typing `popu` the UI suggests `wdt:P1082(population)` if inserting this predicate still would lead to a non-empty query result. This can possibly be achieved by combining the autocompletion in the current QLever-UI that works on FreebaseEasy with mechanism from the EntityFinder introduced in this thesis.

Create Entity Taggings from Wikipedia to Wikidata

Since Wikipedia is a sister project of Wikidata and since both projects are among the most interesting collections of human knowledge that are publicly available it seems somewhat natural to create taggings between these two datasets to be able to also use the SPARQL + Text features of QLever on this combination of datasets.

11.2 Acknowledgements

I want to thank Prof. Dr. Hannah Bast for supervising and reviewing this work and for giving me the great opportunity to present some of the results of this thesis at the DFG meeting in Berlin. I want to thank Niklas Schnelle who currently maintains the QLever codebase and has provided valuable feedback for my changes to the code. I want to thank Florian Kramer who is working on the efficient implementation of the full SPARQL standard for QLever. I want to thank Julian Bürklin and Daniel Kemen. They have developed the QLever-UI which has made working with QLever and evaluating it much more convenient

References

[1] Hannah Bast and Björn Buchhold. "QLever: A Query Engine for Efficient SPARQL+Text Search". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. 2017, pp. 647–656. DOI: 10.1145/3132847.3132921. URL: http://doi.acm.org/10.1145/3132847.3132921.

[2] Hannah Bast and Elmar Haussmann. "More Accurate Question Answering on Freebase". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. Melbourne, Australia: ACM, 2015, pp. 1431–1440. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806472. URL: http://doi.acm.org/10.1145/2806416.2806472.

[3] Rene De La Briandais. "File Searching Using Variable Length Keys". In: *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*. IRE-AIEE-ACM '59 (Western). San Francisco, California: ACM, 1959, pp. 295–298. DOI: 10.1145/1457838.1457895. URL: http://doi.acm.org/10.1145/1457838.1457895.

[4] A. S. Fraenkel, M. Mor, and Y. Perl. "Is text compression by prefixes and suffixes practical?" In: *Acta Informatica* 20.4 (Dec. 1983), pp. 371–389. ISSN: 1432-0525. DOI: 10.1007/BF00264280. URL: https://doi.org/10.1007/BF00264280.

[5] Miguel A. Martinez-Prieto et al. "Practical Compressed String Dictionaries". In: *Inf. Syst.* 56.C (Mar. 2016), pp. 73–108. ISSN: 0306-4379. DOI: 10.1016/j.is.2015.08.008. URL: https://doi.org/10.1016/j.is.2015.08.008.

[6] Donald R. Morrison. "PATRICIA&Mdash;Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4 (Oct. 1968), pp. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481. URL: http://doi.acm.org/10.1145/321479.321481.

[7] Ingo Müller et al. *Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.*

[8] T. Radhakrishnan. "Selection of prefix and postfix word fragments for data compression". In: *Information Processing and Management* 14.2 (1978), pp. 97–106.

Web Ressources

QLever

- https://github.com/ad-freiburg/QLever (source code)
- http://qlever.informatik.uni-freiburg.de (UI/ running instances)

Wikidata

- https://www.wikidata.org/ (main page)
- https://query.wikidata.org (Wikidata query service)

RDF and SPARQL

- https://www.w3.org/RDF/
- https://www.w3.org/TR/n-triples/ (N-Triples format)
- https://www.w3.org/TR/turtle/ (Turtle format)
- https://www.w3.org/TR/rdf-sparql-query/ (SPARQL query language)

Attachment A: The Benchmark Queries

This attachement contains the exact queries that have been performed on QLever and the Wikidata Query Service to obtain the results of the evaluation. When using them in QLever we have to prepend the following list of prefixes:

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wds: <http://www.wikidata.org/entity/statement/>
PREFIX psv: <http://www.wikidata.org/prop/statement/value/>
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wikibase: <http://wikiba.se/ontology-beta#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX qlInt: <http://http://qlever.informatik.uni-freiburg.de/>
```

This step is not necessary for the Wikidata Query Service since it automatically uses those prefixes.

Q1: All Entities Connected to Douglas Adams

QLever: With Language Filter (ql:langtag)

```
SELECT DISTINCT ?y ?yLabel WHERE {
  wd:Q42 ?x ?y .
  ?y rdfs:label ?yLabel .
  ?yLabel qlInt:predicates/langtag qlInt:entities/@en .
}
```

QLever: With Language Filter (@en@rdfs:label)

```
SELECT DISTINCT ?y ?yLabel WHERE {
  wd:Q42 ?x ?y .
  ?y @en@rdfs:label ?yLabel .
}
```

WQS: With Language Filter

```
SELECT DISTINCT ?y ?yLabel WHERE {
  wd:Q42 ?x ?y .
  ?y rdfs:label ?yLabel .
  FILTER (lang(?yLabel) = "en")
}
```

Without Language Filter

```
SELECT DISTINCT ?y ?yLabel WHERE {
```

```
  wd:Q42 ?x ?y .
  ?y rdfs:label ?yLabel .
}
```

Without Label Resolution

```
SELECT DISTINCT ?y WHERE {
  wd:Q42 ?x ?y .
}
```

Q2: German Cities and their German Names

QLever: With Language Filter (ql:langtag)

```
SELECT ?name ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
  ?city rdfs:label ?name .
  ?name qlInt:predicates/langtag qlInt:entities/@de .
}
ORDER BY DESC(?population)
```

QLever: With Language Filter (@en@rdfs:label)

```
SELECT ?name ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
  ?city @de@rdfs:label ?name .
}
ORDER BY DESC(?population)
```

WQS: With Language Filter

```
SELECT ?cityLabel ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "de". }
}
ORDER BY DESC(?population)
```

WQS: Without Language Filter

```
SELECT ?cityLabel ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
  ?city rdfs:label ?cityLabel .
}
ORDER BY DESC(?population)
```

Without Label Resolution

```
SELECT ?city ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 wd:Q183 .
  ?city wdt:P1082 ?population .
}
ORDER BY DESC(?population)
```

Q3: Persons and their Birth Date

QLever: With Language Filter (ql:langtag)

```
SELECT ?person_id ?person_label ?date_of_birth
WHERE {
  ?person_id p:P569 ?date_of_birth_statement .
  ?person_id rdfs:label ?person_label .
  ?person_label qlInt:predicates/langtag qlInt:entities/@en .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth
}
```

QLever: With Language Filter (@en@rdfs:label)

```
SELECT ?person_id ?person_label ?date_of_birth
WHERE {
  ?person_id p:P569 ?date_of_birth_statement .
  ?person_id @en@rdfs:label ?person_label .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth
}
```

WQS: With Language Filter

```
SELECT ?person ?personLabel ?date_of_birth
WHERE {
  ?person p:P569 ?date_of_birth_statement .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
```

WQS: Without Language Filter

```
SELECT ?person ?personLabel ?date_of_birth
WHERE {
  ?person p:P569 ?date_of_birth_statement .
  ?person rdfs:label ?personLabel .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth
}
```

WQS: Without Label Resolution

```
SELECT ?person ?date_of_birth
WHERE {
  ?person p:P569 ?date_of_birth_statement .
  ?date_of_birth_statement psv:P569 ?date_of_birth_value .
  ?date_of_birth_value wikibase:timePrecision "9"^^xsd:integer .
  ?date_of_birth_value wikibase:timeValue ?date_of_birth
}
```

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 also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

_____	_____
Place, date	Signature