Bachelor's Thesis

Efficient Keyword Search For The QLever SPARQL Engine

Nick Göckel

Examiner: Prof. Dr. Hannah Bast
Advisers: Johannes Kalmbach

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

May 4th, 2024

Writing period

05. 01. 2024 – 04. 04. 2024

Examiner

Prof. Dr. Hannah Bast

Advisers

Johannes Kalmbach

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

QLever is an efficient SPARQL search engine that supports combined search on an RDF knowledge base and a collection of texts. In this thesis, we present a new implementation of text search for the QLever SPARQL engine, that simplifies the code and makes future improvements easier. We also present a new feature for QLever, that allows the user to retrieve the completed keywords for prefix keyword searches.

Contents

List of Figures

1 Introduction

In this chapter, we will give a brief introduction to the topic and goal of this thesis. We will start by giving some background information.

A simple and efficient way to store structured data is the RDF (Resource Description Framework) data model. The RDF data model stores data in triples. A triple consists of a subject, a predicate, and an object. Figure 1 shows an example of how the awards won by some famous scientists could be stored in the RDF data model.

Subject	Predicate	Object
<Albert_Einstein>	<Award_Won>	<Nobel_Prize_in_Physics>
<Carl_Bosch>	<Award_Won>	<Nobel_Prize_in_Chemistry>
<Charles_Darwin>	<Award_Won>	<Royal_Medal>
<Marie_Curie>	<Award_Won>	<Nobel_Prize_in_Chemistry>
<Marie_Curie>	<Award_Won>	<Nobel_Prize_in_Physics>

Figure 1: Example table of RDF triples. The Figure shows one possible way to store the awards won by some famous scientists in the RDF data model.

The quasi-standard query language for RDF knowledge bases is SPARQL. With SPARQL it is possible to retrieve and manipulate data stored in the RDF format. Figure 2 shows a simple SPARQL query with its result.

SPARQL queries can be executed with a SPARQL engine. If we want to perform text search as well as look for specific keywords or entities in the text, it is best to use a SPARQL engine that has the extra feature of supporting combined search. This means that the engine is using structured knowledge from an RDF knowledge base and textual information from a collection of texts. Such an engine allows for queries that access and combine information from both sources. Text search in general is possible with plain SPARQL engines. But compared to a SPARQL engine that supports combined search, the text search is less efficient and less user-friendly [1].

```
SELECT ?scientist WHERE {
    ?scientist <Award_Won> <Nobel_Prize_in_Physics> .
}
```

?scientist
<Albert_Einstein>
<Marie_Curie>

Figure 2: Example SPARQL query with its result. This query retrieves all scientists who have won a Nobel Prize in Physics. The table shows the results for the knowledge base described in Figure 1.

This is where the QLever SPARQL engine comes into play. The QLever engine is an efficient SPARQL search engine that also supports combined search. Figure 3 shows an example of a SPARQL+Text query with its result.

```
SELECT * WHERE {
    ?scientist <Award_Won> <Nobel_Prize_in_Physics> .
    ?text ql:contains-entity ?scientist .
    ?text ql:contains-word "radio*"
}
```

?scientist	?text
<Marie_Curie>	Marie Curie says that she and her husband had traded ideas with Sagnac around the time of the discovery of radioactivity.

Figure 3: Example SPARQL+Text query with its result. This query retrieves all scientists who have won a Nobel Prize in Physics and are mentioned in a text that also contains the prefix "radio". The table shows the result for the knowledge base shown in Figure 1 and for a collection of texts that is not explicitly shown here.

1.1 Problem Definition

The QLever SPARQL engine initially performed keyword searches in an efficient but code-wise complex and hard-to-maintain manner. Besides that, the engine could not return the completed keywords when performing prefix keyword searches (as shown in Figure 4).
In this thesis, we will introduce an approach that solves these problems. The approach still performs efficient keyword searches but makes use of an easy and maintainable code structure. On top of that the engine is also now able to return the completed keywords for prefix keyword searches. This makes a whole new set of queries possible and allows for many new applications. For example, with this feature, QLever can be used in a search application where the user can run prefix queries and select for which completed keywords he wants to see the results of and for which not. The feature also makes it possible to group results by the completed keywords and rank these groups by the number of results they contain. Another application could be to use the completed keywords to perform a sort of auto-completion in a search application. Suggesting the user possible keywords that he could be looking for. Figure 4 shows an example of the new feature.

?scientist	**?text**	**?ql_matchingword_text_radio**
<Marie_Curie>	Marie Curie says that she and her husband had traded ideas with Sagnac around the time of the discovery of radioactivity.	radioactivity

Figure 4: Example SPARQL+text query with the completed word feature and the result. This is the same query as in Figure 3, but the result now includes the completed word feature as described above.

2 Background

In this chapter, we will give a brief introduction to the background of the topics addressed in this thesis and acquire the necessary knowledge.

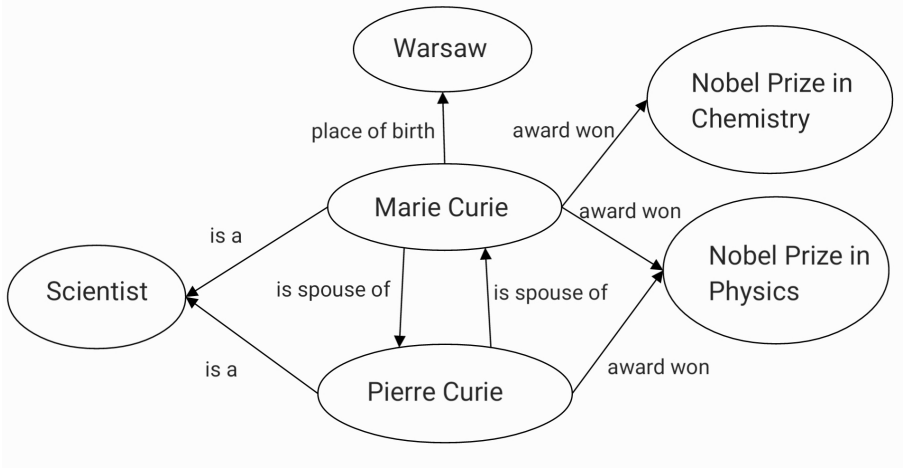
2.1 Resource Description Framework

The Resource Description Framework (RDF) is a data model standardized by the World Wide Web Consortium (W3C). It is used to represent and interchange information in an unambiguous manner. In the RDF data model, information is represented in triples. An RDF knowledge graph is a set of triples. A triple consists of a subject, a predicate, and an object. While the subject and the object represent things or concepts the predicate represents a relation between them. The RDF data model can be seen as a graph of triples, where the subject and object are nodes and the predicate is an edge between them (as shown in Figures 5 and 6).

Figure 5: Basic RDF triple as a graph.

The RDF is often used because it is a simple data model with high expressive power. It can describe resources and their relations in a structured way.

Web Sources:
`https://www.w3.org/RDF/`
`https://www.w3.org/TR/rdf11-concepts/`



Subject	Predicate	Object
<Marie_Curie>	<born_in>	<Warsaw>
<Marie_Curie>	<is-a>	<Scientist>
<Marie_Curie>	<Award_Won>	<Nobel_Prize_in_Chemistry>
<Marie_Curie>	<Award_Won>	<Nobel_Prize_in_Physics>
<Marie_Curie>	<is_spouse_of>	<Pierre_Curie>
<Pierre_Curie>	<is-a>	<Scientist>
<Pierre_Curie>	<Award_Won>	<Nobel_Prize_in_Physics>
<Pierre_Curie>	<is_spouse_of>	<Marie_Curie>

Figure 6: Example of a small RDF knowledge base. Once depicted as a graph and once as a table. Both representations show the same information.

2.2 SPARQL

SPARQL is the most commonly used query language for RDF knowledge graphs. It is specified by the W3 Consortium. SPARQL can retrieve and manipulate data stored in the RDF format. The query structure is similar to SQL. The two most important keywords are SELECT and WHERE. In the SELECT clause, it is specified which variables should be returned. Note that in SPARQL variables start with a question mark. The WHERE clause usually consists of one or more triples. Each triple specifies a semantic relation between the subject and the object. The subject and object can be a variable or refer to specific things. Figure 2 shows a simple SPARQL query, where only the variable `?scientist` is selected. In this query, the WHERE clause only contains one triple in which the subject is a variable and the object is a specific entity. If there are multiple triples, the results of each triple

are joined together like a logical AND on columns with equal variables. Figure 7 shows an example where the partial results of two triples are joined together on the common variable `?scientist`. So as a result, only the scientists that won a Nobel Prize in Physics AND Chemistry are returned.

```
SELECT * WHERE {
    ?scientist <Award_Won> <Nobel_Prize_in_Physics> .
    ?scientist <Award_Won> <Nobel_Prize_in_Chemistry> .
}
```

?scientist
<Marie_Curie>

Figure 7: Example SPARQL query with its result. This query retrieves all scientists who have won a Nobel Prize in Physics and Chemistry. The table shows the results for the knowledge base described in Figure 1.

If we change the query so that there are no two same variables and therefore no common column to join on, the result would be created by calculating the cartesian product of the two partial results.
Figure 8 shows an example where the partial results of two triples are not joined together because no common variable exists. As a result, we get all combinations of a scientist who won a Nobel Prize in Physics and a scientist who won a Nobel Prize in Chemistry.

Web Sources:
`https://www.w3.org/TR/sparql11-query/`

```
SELECT * WHERE {
    ?scientist1 <Award_Won> <Nobel_Prize_in_Physics> .
    ?scientist2 <Award_Won> <Nobel_Prize_in_Chemistry> .
}
```

?scientist1	?scientist2
<Albert_Einstein>	<Carl_Bosch>
<Albert_Einstein>	<Marie_Curie>
<Mari_Curie>	<Carl_Bosch>
<Mari_Curie>	<Marie_Curie>

Figure 8: Example SPARQL query with its result. This query retrieves all possible combinations of a scientist who has won a Nobel Prize in Physics and a scientist who has won a Nobel Prize in Chemistry. The table shows the results for the knowledge base described in Figure 1.

2.3 QLever

QLever is an efficient SPARQL+Text search engine. It is being developed by the Chair of Algorithms and Data Structures at the University of Freiburg. What sets QLever, and other SPARQL+Text search engines, apart from a standard SPARQL engine is that they support combined search with structured knowledge from an RDF graph and textual information from a collection of texts. This combined search enables users to access a broad range of information in a single query. For this feature, the QLever engine has added two unique predicates, ql:contains-word and ql:contains-entity. The predicate ql:contains-word can be used to search for a specific word or prefix in the text collection. The predicate ql:contains-entity can be used to search for entities in the text collection. Entities can be objects, people, or other things that are referred to in the text.
This enables QLever to run queries like the one shown in Figure 3.
We will examine the text index, a key part of the QLever engine, in more detail in Chapter 3. But for those interested a more detailed description of the QLever engine can be found in the 2017 paper "QLever: A Query Engine for Efficient SPARQL+Text Search" by Bast and Buchhold [2].

3 QLever Index Structure

In this chapter, we will give a brief introduction to the structure of the QLever index. QLever has two indices: a knowledge base index and a text index. The knowledge base index stores the standard RDF relation triples. The text index stores text records as well as the words and the entities they contain. For this thesis, we will only need to further inspect the text index.

But before we start we will introduce a few basic words that are used in the context of the QLever text index.

Entity is a word that is used in the context of a search. It can be a person, a place, or a thing.

Term is a word that is used in the context of a search. It can be a word or a prefix.

Text record is a text that is stored in the text index. For the QLever engine, it corresponds most of the time to a sentence.

Vocabulary is a list of all text records, words, and entities that are stored in the text index. It assigns each text record, word, and entity a unique identifier.

Text record ID is a unique identifier for a text record in the vocabulary.

Word ID is a unique identifier for a word in the vocabulary.

Entity ID is a unique identifier for an entity in the vocabulary.

Score is a value that represents the relevance of an entity in a text record. It can be used to rank the results of a query.

Word posting is a list containing a text record ID, a word ID, and a score, where the word contains the term and the text record contains the word.

Entity posting is a list containing a text record ID, an entity ID, and a score, where the text record contains the entity.

3.1 Text Index

The QLever text index consists of two lists for every four-letter prefix occurring in any saved text [2]. One list contains word postings and one list contains entity postings.
The first list, the context list, contains the word postings that fit the prefix. This means that the list contains a triple of text record ID, word ID, and score for each occurrence of a word containing the prefix in a text record. So there is one word posting for each word containing the prefix, in each text record [3]. Note that the prefix is always contained in the word of all word posting in its list.
The second list, the entity context list, contains the entity postings that fit the prefix. This means that the list contains a triple of text record ID, entity ID, and score for each occurrence of an entity in a text record, where the text record contains the prefix. So there is one entity posting for each entity in each text record containing the prefix. This means if a text record contains multiple entities, there will be multiple entity postings containing the same text record ID [3].
Note that this index structure yields the problem that prefix search with prefixes of length less than four is not supported. This is because the result of a prefix search with a prefix of length less than four is spread out over multiple lists. For prefixes of length longer than four there is some computation necessary to get the right results. This computation is explained in Subsection 4.2.1 in more detail.
Figure 9 shows an example of the text index structure for the prefix rela*. Figure 10 shows to what words, entities or sentences the IDs in the example correspond.

This index structure allows for fast prefix search as well as quick and independent access to word and entity postings.

Note: Figure 10 shows the list of IDs used in this example and what they mean.

Context list for prefix rela*:

Text record IDs:	...	14	14	15	16	17	18	...
Word IDs:	...	719	741	722	719	778	741	...
Scores:	...	1	1	1	1	1	1	...

Entity context list for prefix rela*:

Text record IDs:	...	14	15	16	17	18	18	...
Entity IDs:	...	12	18	15	26	18	12	...
Scores:	...	1	1	1	100	100	1	...

Figure 9: Example of the text index structure for the prefix rela*. Here one column corresponds to one word/entity posting.

We assume the following IDs:

Text record ID 14: In 1908 he published the first of several papers on relativity, in which he derived the mass-energy relationship in a different way from Albert Einstein's derivation.

Text record ID 15: He discussed the "principle of relative motion" in two papers in 1900.

(Although only implicitly mentioned with the pronoun "he", the entity here is Hendrik Lorentz)

Text record ID 16: This relationship became known as Haber's rule.

Text record ID 17: Specifically, Losev patented the "Light Relay" and foresaw its use in telecommunications.

Text record ID 18: Lorentz published a series of papers dealing with what he called "Einstein's principle of relativity".

Entity ID 12: Albert Einstein
Entity ID 15: Fritz Haber
Entity ID 18: Hendrik Lorentz
Entity ID 26: Oleg Losev

Word ID 719: relationship
Word ID 722: relative
Word ID 741: relativity
Word ID 778: relay

Figure 10: List of IDs used in the examples.

4 Approach

In this chapter, we will explain our approach to how we improved the QLever text search. Briefly summarized we improved it by restructuring parts of the code and adding the previously described feature of returning completed keywords for prefix searches. But first, we will take a look at a research prototype we created and what was wrong with it. Then we will go on and explain the final version that we implemented.

4.1 Research Prototype

As a first measure to implement the keyword completion feature, we created a research prototype. This prototype just extended the already existing text search with the added feature of returning completed keywords.
Initially, there was in QLever a whole separate code section just to perform the text search. The section had its own methods for retrieving text record IDs and entity IDs from the text index as well as merging, filtering, and sorting the results.
To implement the feature of returning completed keywords for prefix searches we had to update the code in multiple places. First and foremost we now also needed to retrieve the word IDs from the text index. But besides that, we also needed to update a big part of the code so that the merging and filtering methods would handle the word IDs properly.
We soon realized that this text search section had a lot of code duplication and that there was a much simpler way to implement the text search. So we decided to restructure the code and implement the text search in a new way. The new approach is thoroughly described in the next section. More extensive reasoning on the advantages of the restructuring can be found in Chapter 5.

4.2 Final Implementation

In the final implementation, we restructured the code and realized the text search in a new way. For this implementation, we made use of another important part of QLever, the query planner. The query planner lies at the very heart of the new restructured implementation. What the query planner does is it creates and optimizes the way and order operations are executed. Operations are the basic building blocks of a query and can be functions like reading from the index, filtering, joining, and sorting. There are many different ways to execute a query. The query planner decides which way is the most efficient [1].
A big advantage of this new implementation is that the query planner is given more freedom. Because the text search is now split into multiple smaller operations instead of just one big operation. So the query planner can now plan and optimize the tasks that were previously done in the text search section.
This approach also allows the query planner itself to handle many tasks that were previously done by algorithms specifically designed for just a single use case in the text search. This increase in simplicity and maintainability is a major reason why we decided to implement this approach.
In this implementation, text search is realized through the following three operations:

The **TextIndexScanForWord** operation finds all text records that contain a user-specified term. If the term is a prefix the operation also returns the word that the prefix was completed to. For each word in a ql:contains-word statement one TextIndexScanForWord operation is created.

The **TextIndexScanForEntity** operation finds all text records and the entities they contain for a specific term. The term is not specified by the user but is chosen by the query planner. The query planner chooses the term out of all terms that are specified in a ql:contains-word statement. The structure of the text index makes this term necessary. This also means that a ql:contains-entity statement can never be used without a ql:contains-word statement in the same query. The operation can also be called with a by the user-specified fixed entity. If this is the case the operation returns every text record that contains the specified entity and the term. For each ql:contains-entity statement one TextIndexScanForEntity operation is created.

The **TextLimit** operation is used to limit the number of text records that appear per entity. For each text variable in the query, one TextLimit operation

is created. A text variable is a variable, that contains text excerpts. The limit is specified by the user. Figure 19 shows an example query tree for the query specified in Figure 11. A query tree is a tree structure that shows the order in which the operations are executed. The leaves of the tree are executed first and the root is executed last.

For example, for the query in Figure 11, two TextIndexScanForEntity, three TextIndexScanForWord and two TextLimit operations would be created.

```
SELECT * WHERE {
    ?scientist <is-a> <Scientist> .
    ?scientist <Gender> <Female> .
    ?text1 ql:contains-entity ?scientist .
    ?text2 ql:contains-entity <Ada_Lovelace> .
    ?text1 ql:contains-word "machine beca*" .
    ?text2 ql:contains-word "fasc*" .
}
TEXTLIMIT 5
```

Figure 11: Example SPARQL query containing multiple ql:contains-word, ql:contains-entity statements and a text limit.

4.2.1 TextIndexScanForWord operation

The TextIndexScanForWord operation reads from the text index all word postings for a user-specified term and writes the result to an ID table. An ID table is a data structure that stores IDs in a table format. These IDs can, for example, represent text records, words, or entities. For this operation, the result ID table has two columns. One column contains the text record IDs and the other column contains the word IDs. If the user-specified term is not a prefix the word ID column is not needed and the ID table only has one column. Every row in the ID table corresponds to one text record containing a word that contains the term.
Figure 12 shows two example calls of the TextIndexScanForWord operation and the resulting ID table. Note how the text record ID column has the name of the text variable `?text`. This is important because it makes merging with other statements possible.

Note: Figure 10 shows the list of IDs used in this example and what they mean.

```
?text ql:contains-word
      "relativ*"
```

Result of operation:

?text	?ql_matchingword _text_relativ
...	...
14	741
15	722
18	741
...	...

```
?text ql:contains-word "relativity"
```

Result of operation:

?text
...
14
18
...

Figure 12: Example calls of TextIndexScanForWord operation with and without prefix.

In the following, we will describe all the important steps of how the TextIndexScanForWord operation retrieves its result. On a high level, what the operation does is it first gets the right context list for the term. Then the operation reads the entries of the context list to an ID table. As an optional last step, it filters out entries with non-fitting words.

Getting the right context list

To get the right context list, the first step is to get the ID range of the term. This ID range consists of the word ID of the first word in the vocabulary that has the prefix and the word ID of the last word in the vocabulary that has the prefix. For a word that is not prefixed the beginning and end of the ID range is just the word ID of the word itself.

With this ID range, we can now find via binary search the first text block that contains the beginning of the ID range. A text block is here a data structure that consists of the context list and the entity context list for a certain four-letter prefix. Four is just a number set by QLever and has no deeper meaning. Note that the found text block should always also contain the end of the ID range. This is because one text block always contains all word postings for the shortest possible prefix (four letters). Figure 13 visualizes the concept of getting the right context list.

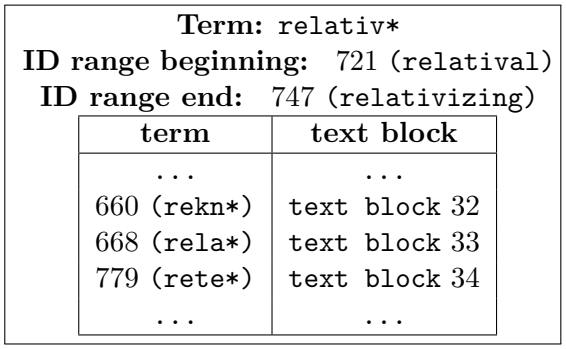


Figure 13: Visualization of how the right text block is found. If we have got the right text block, we can easily get the correct context list from it.

Reading the entries of the context list

The next step is to read the entries of the context list and write them to an ID table. For this, we access the context list of the text block from above. Then we can just read the text record IDs and the word IDs from the context list to a new empty ID table. What we now have is a two-column ID table. One column contains the text record IDs and the other column contains the word IDs. So one entry corresponds to one text record and one word that the text record contains. Figure 14 shows an example of a context list and the resulting ID table.

Filtering out non-fitting words

The last step is to filter out the entries for which the word ID is not in the ID range. There are some cases where this is not necessary. For example, if the user-specified term is a four-letter prefix. But in most cases, it is necessary because after reading the entries of the context list we have a superset of all the entries that contain the term. We have a superset because we always read the whole text block but there are only text blocks for the shortest possible prefix size. So if we have a longer prefix or a non-prefixed term we still get all the entries even though some words do not fit.

We filter out the unfitting entries by iterating over the ID table and checking for each entry if the word ID is in the ID range. If it is not we remove the entry from the ID table. Figure 15 shows an example of an ID table before and after the filtering step.

Note: Figure 10 shows the list of IDs used in this example and what they mean.

Context list for term rela*:

Text record IDs:	...	14	14	15	16	17	18	...
Word IDs:	...	719	741	722	719	778	741	...
Scores:	...	1	1	1	1	1	1	...

Resulting ID table for term rela*:

Text record IDs	Word IDs
...	...
14	719
14	741
15	722
16	719
17	778
18	741
...	...

Figure 14: Example of the context list for the term rela* and the resulting ID table.

Reminder:

Note: Figure 10 also shows the list of IDs used in this example and what they mean.

ID range: 721 (relatival) - 747 (relativizing)

Word IDs: relationship: 719, relative: 722, relativity: 741, relay: 778

Table before filtering:

Text record IDs	Word IDs
...	...
14	719
14	741
15	722
16	719
17	778
18	741
...	...

Table after filtering:

Text record IDs	Word IDs
...	...
14	741
15	722
18	741
...	...

Figure 15: Example of ID table before and after word filtering. All entries where the word ID is not in the ID range were removed. This means that only entries that begin with "realtiv" are kept.

4.2.2 TextIndexScanForEntity operation

The TextIndexScanForEntity operation reads from the text index all entity postings for a given term and writes the result to an ID table. There are two different ways to call the TextIndexScanForEntity operation. One way is to call it with a specific entity, and the other way is to call it with an entity variable. An entity var is a variable, that only contains entities. An example of both calls can be seen in Figure 16.
If the operation is called with a specific entity it returns an ID table with two columns. One column contains the text record IDs and one column contains the scores. Every entry in the ID table corresponds to one text record, that contains the term as well as the specific entity, and the corresponding score.
If the operation is called with an entity variable it returns an ID table with three columns. One column contains the text record IDs, one column contains the entity IDs and one column contains the scores. Every entry in the ID table corresponds to one entity, the text record containing the entity, and the corresponding score.
On which term the operation is called is not directly specified by the user. The query planner decides which term is used by the TextIndexScanForEntity operation. Qualified for that are all terms that are specified in a ql:contains-word statement in the query. From this list, the query planner decides for the term that probably will yield the smallest intermediate result and therefore will be the most efficient to use. For the TextIndexScanForEntity operation, the query planner is also important because by itself the TextIndexScanForEntity operation returns a superset of the actual correct result. Only by merging with the TextIndexScanForWord operation, we can filter out the wrong entries. This merging step is one that was previously, for the research prototype, done manually but is now done by the query planner.
To get a better understanding of the TextIndexScanForEntity operation we will describe in the following all the important steps of how the TextIndexScanForEntity operation retrieves its result. On a high level, the first step is the same as for the TextIndexScanForWord operation. The operation gets the right entity context list for the term. Then it reads the text record IDs, the entity ID, and the scores from the entity context list to a new empty ID table. As an optional last step, if it is wanted by the user, the operation filters out every entry where the entity ID does not match the user-specified fixed entity.

Note: Figure 10 shows the list of IDs used in this example and what they mean.

Example calls of ql:contains-entity first with an entity var then with a fixed entity, as well as the corresponding results produced by the TextIndexScanForEntity operation.

```
?text ql:contains-entity ?scientist .
```

Result of operation:

?text	?scientist	?score
...	...	...
14	12	1
15	18	1
16	15	1
17	26	100
18	18	100
18	12	1
...	...	...

```
?text ql:contains-entity <Albert_Einstein> .
```

Result of operation:

?text	?score
...	...
14	1
18	1
...	...

Figure 16: Example calls of TextIndexScanForEntity operation with and without a fixed entity.

Getting the right entity context list

Getting the entity context list happens in the same manner as the retrieval of the context list described above for the TextIndexScanForWord operation.
Here lies also the reason why we need to use a term for the TextIndexScanForEntity operation even though we just want to read the entities. As described in Chapter 3 the text index has two lists for each four-letter prefix. We can find the entities in one of these lists. So to access the entities we need to get the right entity context list and for this, we need the term.

Reading the entries of the entity context list

The next step is to read the entries to an ID table. For this, we access the entity context list of the text block we got from the previous step. Then we can just read the text record IDs, the entity IDs, and the scores from the entity context list to a new empty ID table. Figure 17 shows an example of an entity context list and the resulting ID table.

Note: Figure 10 shows the list of IDs used in this example and what they mean.

Entity context list for term rela*:

Text record IDs:	...	14	15	16	17	18	18	...
Entity IDs:	...	12	18	15	26	18	12	...
Scores:	...	1	1	1	100	100	1	...

Resulting ID table for term rela*:

Text record IDs	Entity IDs	Scores
...	...	...
14	12	1
15	18	1
16	15	1
17	26	100
18	18	100
18	12	1
...	...	...

Figure 17: Example of the entity context list for the term rela* and the resulting ID table.

Filtering out non-fitting entities

If the user specified a fixed entity, the last step is to filter out the entries where the entity ID does not correspond to the specified fixed entity. We do this by getting the index of the fixed entity in the vocabulary. Then we iterate through the ID table and check for each entry if the index of the entity of the current entry and the index of the fixed entity are the same. If they are not we remove the entry from the ID table. So we end up with an ID table that is a subset of the input ID table which only contains the entries that contain the fixed entity specified by the user. Figure 18 shows an example of an ID table before and after the filtering step.

Fixed entity: 12 (Albert Einstein)
Note: Figure 10 also shows the list of IDs used in this example and what they mean.

Table before filtering:

Text record IDs	Entity IDs	Scores
...	...	...
14	12	1
15	18	1
16	15	1
17	26	100
18	18	100
18	12	1
...	...	...

Table after filtering:

Text record IDs	Entity IDs	Scores
...	...	...
14	12	1
18	12	1
...	...	...

Figure 18: Example of ID table before and after entity filtering. All entries where the entity ID is not 12 were removed.

4.2.3 TextLimit operation

The TextLimit operation is used to limit the number of text records that appear per entity. It gets an ID table as its input and returns an ID table that only contains the text records with the k highest scores for each entity. Where k is a number specified by the user. If there are multiple entities for one text variable the operation only keeps the first k text records for each entity combination.
Currently, the text limit can only be set globally, for the whole query, by the user. The TextLimit operation on the contrary needs to be applied once for every text variable in the query. Each operation takes all to the text variable corresponding entities and scores into account. The TextLimit operations can be applied at different points in the query tree. The query planner decides where the TextLimit operation is applied so that the query is as efficient as possible. Figure 19 shows an example query tree with two TextLimit operations.

On a high level, the operation just sorts the ID table and then iterates through the ID table and for each entity combination only keeps the first k text records. Figure 20 shows an example of an ID table before, during, and after the TextLimit operation for a text limit of 1.

Figure 19: Example query tree for the query specified in Figure 11. TextIndexScanForEntity operations are shown in green, TextIndexScanForWord operations are shown in blue and TextLimit operations are shown in orange.

Example of an ID table before, during and after the TextLimit operation for a text limit of 1. Note that the IDs in this example are purely random and are not related to the IDs used in the rest of this thesis.

ID table before TextLimit operation:

TR IDs	E1 IDs	E2 IDs	E3 IDs	W IDs	Score1	Score2	Score3
5	1	1	2	1	1	1	2
4	0	3	1	1	7	7	7
5	5	23	17	2	6	6	4
7	1	1	1	1	2	21	2
2	0	3	1	0	2	1	1
2	5	9	5	0	5	2	6
19	1	1	1	4	22	2	1

ID table after it was sorted
(this is actually during the TextLimit operation):

It is sorted in descending order first on the Entity1 column, then the Entity2, then the Entity3, then on the sum of the three score columns and finally on the Text record column.

TR IDs	E1 IDs	E2 IDs	E3 IDs	W IDs	Score1	Score2	Score3
5	5	23	17	2	6	6	4
2	5	9	5	0	5	2	6
5	1	1	2	1	1	1	2
19	1	1	1	4	22	2	1
7	1	1	1	1	2	21	2
4	0	3	1	1	7	7	7
2	0	3	1	0	2	1	1

ID table after the TextLimit operation:

TR IDs	E1 IDs	E2 IDs	E3 IDs	W IDs	Score1	Score2	Score3
5	5	23	17	2	6	6	4
2	5	9	5	0	5	2	6
5	1	1	2	1	1	1	2
4	0	3	1	1	7	7	7

Figure 20: Example of an ID table before, during and after the TextLimit operation. In the tables above, TR IDs are Text Record IDs, E1 IDs are Entity 1 IDs, E2 IDs are Entity 2 IDs, E3 IDs are Entity 3 IDs, W IDs are Word IDs and the Score columns are the scores for the corresponding entities.

5 Evaluation

In this chapter, we will evaluate the restructuring of the text search and the implementation of the completed word feature for prefix text searches. We will mainly focus on the functionality and the code quality of the implementation, but will also compare the new text search implementation with the old one. A meaningful evaluation of the new feature is only to a limited extent possible, as it is a new feature and there is no previous implementation to compare it to.

5.1 Feature Evaluation

The new feature of returning the completed words for prefix searches is entirely implemented and works with full functionality as imagined. It is now possible to perform a prefix text search and get for each entry in the result the word the prefix was completed to. Besides that, we also implemented a new feature, which allows the user to search for fixed entities in the text collection. On top of that, we were also able to fix some existing bugs, which were related to the text search. For example, queries using multiple ql:contains-entity statements on the same text variable used to return an error. These queries now work as expected.

5.2 Code Quality

When it comes to the code quality of the implementation we were able to comply with the high standards of the QLever code base. We achieved this by doing multiple rounds of intensive peer review and using analysis tools like Sonarqube.
With the restructuring of the text search, we were even able to greatly improve the quality of this part of the code. A reason for this is that we implemented the reading of the text index in a new way, that uses already existing code and is already optimized and tested. Because of this, we were also able to reduce code duplication and shorten the code by a few thousand lines. This also greatly improved the maintainability and readability of the code. One of the many challenges of this

thesis was to understand the at the beginning very complex part of the code base that reads the text index. We believe that by deleting duplication and relying on already existing well-maintained code we were able to increase the readability and maintainability of the code.

To ensure this high quality of the code we thoroughly tested the implementation using unit tests and end-to-end tests. We also used a code coverage tool to ensure that all important parts of the code are tested.

6 Future Work

In this chapter, we will talk about possible future work that can be done to improve this particular feature or the QLever engine in general.

We have noticed that the text index and its implementation can be a bottleneck when it comes to what the QLever engine can do. It makes it very hard for some features to be implemented. For example running a query, that has a ql:contains-entity statement but no ql:contains-word statement. Or running a query with a prefix shorter than four letters. Those limitations could be fixed by restructuring the text index itself. There are many different ways to do this. One way would be to truly use separate indices for the words and the entities. On top of that, the text index could be realized by storing the word postings in a list sorted by the word ID. This would allow the user to find text excerpts containing prefixes of any length.
In general, there are multiple ways to restructure the text index which all could benefit the QLever engine in different ways. Because we significantly improved the code quality of the code section that reads from the text index, it now is easier than ever to implement the text index in a new way.

One popular type of query that is notoriously hard to program in SPARQL is the so-called "argmax" query. This is a query that returns the entity with the highest value of a certain property. It gets even harder when we want to return the top k entities. This is a query that is very hard to implement in SPARQL and therefore also in QLever. But with the text limit, we got an operation that does something very similar. It returns the text records with the k highest scores for each entity. So with just a bit of generalization, the text limit could be used as an argmax operation. This could be a very powerful feature that could vastly improve the QLever engine.

7 Acknowledgements

First and foremost, I would like to thank my advisor Johannes Kalmbach for his guidance and support throughout the writing of this thesis. He was always available and took all the time I needed to help me whenever I had questions. Thanks to him I was able to learn a lot. I would also like to thank Prof. Dr. Hannah Bast for supervising this thesis. Of course, I would also like to thank my family and friends for their support and encouragement. And also everyone who took the time and effort to proofread this thesis.

Bibliography

[1] H. Bast, J. Kalmbach, T. Klumpp, and C. Korzen, "Knowledge graphs," Book chapter in an unpublished book, 2023. [Online]. Available: https://ad-publications.cs.uni-freiburg.de/CHAPTER_knowledge_graphs_BKKK_2023.pdf.

[2] H. Bast and B. Buchhold, "Qlever: A query engine for efficient sparql+text search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, Singapore, Singapore: Association for Computing Machinery, 2017, pp. 647–656, ISBN: 9781450349185. DOI: 10.1145/3132847.3132921. [Online]. Available: https://ad-publications.informatik.uni-freiburg.de/CIKM_qlever_BB_2017.pdf.

[3] H. Bast and B. Buchhold, "An index for efficient semantic full-text search," in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, ser. CIKM '13, San Francisco, California, USA: Association for Computing Machinery, 2013, pp. 369–378, ISBN: 9781450322638. DOI: 10.1145/2505515.2505689. [Online]. Available: https://ad-publications.informatik.uni-freiburg.de/CIKM_broindex_BB_2013.pdf.