Bachelor Thesis

# Improving the full-text index of QLever

Felix Meisen

| | |
|---|---|
| Examiner: | Prof. Dr. Hannah Bast |
| Supervisor: | Johannes Kalmbach |

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

2025-09-11

## 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.

Bad Krozingen, 11.09.2025
Place, date

[illegible]
Signature

# Abstract

QLever is a SPARQL+Text engine that can be used to build and query an index of a knowledge base and a text corpus. QLever extends SPARQL by two QLever-specific predicates for text search. Those predicates are *ql:contains-word* and *ql:contains-entity* [5]. While the predicate *ql:contains-word* can be used to retrieve texts containing certain words using a *WordScan*, the predicate *ql:contains-entity* returns entities co-occurring in texts together with words using an *EntityScan*. This allows for a "combined search on a knowledge base and a text corpus, in which named entities from the knowledge base have been identified" [5].

This work implements three main changes, improving the full-text index of QLever. The first change modifies the way the text entities of the knowledge base, so-called literals, are added to the full-text index. QLever already provided an option to add all literals to the full-text index, but when querying a full-text index built from all literals, the corresponding literals were not shown in the result. Furthermore, adding all literals leads to many uninformative literals being part of the full-text index. With the change presented in this work, the user can apply a filter to the triples containing literals during index building. Choosing the right filter can lead to a smaller full-text index without loosing query result quality, potentially even improving it. Additionally, the literals are now shown in the results.

The second change allows the user to use a simpler format for the text corpus. A simple text corpus only consists of one file containing all documents together with an identifier for each document. For QLever, the `docsfile` is exactly this, but QLever expected a `docsfile` and a so-called `wordsfile`. This `wordsfile` essentially contains all words and entities appearing in the documents of the `docsfile`. Since the `wordsfile` contains words from multiple so-called text records that are overlapping subsets of documents, the creation of your own `wordsfile` can be difficult. The reason documents were split up into these text records is the automated identification of entities in documents [6]. This logic is complicated and not easily understood. Therefore, the option to only use the simple `docsfile` for full-text index building was added. To still keep the functionality of *ql:contains-entity*, another option was added that only adds the entities of the `wordsfile` to the full-text index. This leads to similar functionality with a much simpler input.

The third and last change simplifies the underlying full-text index structure and provides an option to choose the block size of the full-text index. More importantly, the change

also removes the fixed prefix length of the *WordScan*. This means that whereas before the user could only use *WordScan*s with prefixes having a minimum length of four, they can now use prefixes of arbitrary length like "com*" or even "*".

# Contents

# 1. Introduction

Over the years of human existence, one of the most important tools was, is, and will be information. The amount of information accessible is growing steadily. Growing with it is the challenge to find certain information in this mass. Database engines try to solve this problem. For them to work, the data they operate on needs to have a specific form. One form is a graph database. These databases consist of entities linked together by directional relations. One common way to model such a database is the so-called **R**esource **D**escription **F**ramework, or short RDF [12]. In the RDF model there are only triples. Those triples have a subject, predicate, and object. We can think of the subject and object as nodes in a graph, while the predicates, define the relation. The direction is going from subject to object. There are three different types of RDF entities [12]. The most common type is an **I**nternational **R**esource **I**dentifier, short IRI. As said in the name, IRIs are used to assign entities a unique identifier. IRIs are also used for predicates defining relations. The second most common type is a literal. Literals can either consist of strings or of an IRI and a corresponding value. An example of the latter would be "3375222"^^⟨http://www.w3.org/2001/XMLSchema#int⟩. The first part is the number "3375222", whereas the second part is the IRI defining the number as an integer. The third type of RDF entities are blank nodes. Blank nodes can be used internally by RDF engines to extend the functionality of the database. They are local identifiers and are disjoint from IRIs and literals. An example of a database consisting only of simple IRIs can be seen in Figure 1.1. RDF databases can be queried using the **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage, or SPARQL for short. A simple SPARQL query can be seen in Figure 1.2, and the matching result in Figure 1.3.

Besides so-called structured data like RDF databases, there exists unstructured data like text. A collection of texts assigned with an ID can also be considered a database. For an engine working on a text database, a common goal is to find all documents containing a certain word or prefix. One thing RDF and text datasets have in common is querying the dataset without preprocessing is slow. Therefore, an index has to be built. An everyday example of an index would be a library. It contains all the data and maintains the information on where to find that data. This reduces the time needed to find certain information. Because of the different nature of datasets, most database engines only allow for one form of data. However, QLever is a SPARQL+Text engine that can be used to build and query indexes built from RDF as well as text datasets, even linking them together.

As explained above, QLever first needs to build an index for the RDF database as well

Example triples from *Freebase easy*:

| | | |
|---|---|---|
| <Barack_Obama> | <is-a> | <Person> |
| <Barack_Obama> | <Profession> | <Politician> |
| <Barack_Obama> | <Place_of_birth> | <Honolulu> |
| <London> | <is-a> | <Olympic_host_city> |
| <London> | <Contained_by> | <Great_Britain> |
| <Leonardo_da_Vinci> | <is-a> | <Artist> |
| <Leonardo_da_Vinci> | <Profession> | <Inventor> |
| <Mona_Lisa> | <Artist> | <Leonardo_da_Vinci> |
| <Mona_Lisa> | <Art_Form> | <Painting> |
| <Google> | <is-a> | <Brand> |
| <Google> | <Peer> | <Larry_Page> |

Figure 1.1: Excerpt of the *Freebase easy* knowledge base showing the RDF scheme

Simple SPARQL query:

```
SELECT * WHERE {
    <Barack_Obama> ?p ?o .
}
```

Figure 1.2: A simple SPARQL query that can be used to find information about the entity `<Barack_Obama>`.

as for the text database. Afterward, the user can start a QLever server and run queries on it. An N-Triples, N-Quads, or Turtle file is used to build the RDF index. The full-text index can be built from three main sources. The simplest source is all literals of the RDF database. The more complicated source is a so-called `wordsfile` and a respective `docsfile`. The `docsfile` provides all documents that should be added to the full-text index as well as a unique ID for them. The `wordsfile` not only contains the important words of the documents but also all entity mentions in said documents. These entity mentions directly correspond to entities in the RDF database. This means QLever can be used to combine text search with classic SPARQL queries to scan whether an entity is mentioned in a text together with a word or prefix.

As mentioned in the abstract, this work discusses multiple changes made to the full-text part of QLever. The first change modifies and extends the full-text index building from literals. Literals belonging to the full-text index are now pre-saved. This leads to them now being shown in a query result instead of returning empty results. Also, a regex filter was implemented. This filter can be used to specify what predicates hint at literal

Simple SPARQL query result:

| ?p | ?o |
|---|---|
| <is-a> | <Person> |
| <Profession> | <Politician> |
| <Place_of_birth> | <Honolulu> |

Figure 1.3: Results of the simple SPARQL query from Figure 1.2 executed on the small excerpt of the *Freebase easy* dataset seen in Figure 1.1.

objects that should be added to the full-text index. For example, the user can whitelist the predicate `<has-description>` and all literal objects following said predicate are added to the full-text index.

The second change implemented in this work enables using only the `docsfile` to build the full-text index. Together with this, the possibility to use the `wordsfile` only for the entity mentions was added. This leads to the building of the full-text index requiring less understanding about how QLever internally works and improving the result quality.

The third change implemented modifies the full-text index structure. To build the full-text index QLever saves information for each word and entity occurring in a text record to disk. This information consists of a `TextRecordIndex` specifying which text the word or entity belongs to, a `WordVocabIndex` or `VocabIndex`, which are internal identifiers for words and entities, and a `Score`. Each combination of information is called a posting and has its own entry. These postings are sorted into blocks. Before this work each block contained all postings for all words starting with the same four letter prefix. For example, the block "test*" contained postings for "testing" as well as "testosterone" and so on. This leads to a remarkable difference in block size depending on how often a prefix occurs in the text dataset. In theory, the different block sizes lead to uneven query times depending on what block is retrieved. Now users can choose the block size freely. Together with this change came an even more important change that improves the retrieval. Before this, only one single block could be returned by a *WordScan*. This meant there was a minimal prefix length of four. Now multiple blocks can be read and merged, making it possible to execute *WordScan*s using prefixes like "com*" or even "*".

We will first briefly look at similar engines providing a combination of SPARQL and text search. Then, to understand and evaluate all three changes, the full-text index building of the old QLever version is explained. Together with this, the *WordScan* and *EntityScan* are explained. Afterward, the theoretical analysis of the full-text index building, and of the retrieval is shown. Following this, each feature is explained individually. To see how the changes affect index building times and query results an empirical analysis is shown and the results of it are discusses. At the end, an outlook is given on how to further improve the individual features and the full-text index of QLever.

Three publicly available datasets are used in this work. The scientists dataset available at `https://github.com/ad-freiburg/qlever/raw/master/e2e/scientist-collection.zip` (last accessed 2025-09-10). The freebase easy dataset available at `https://freebase-easy.cs.uni-freiburg.de/dump/` (last accessed 2025-09-10). And the yago-3 dataset available at `https://yago-knowledge.org/downloads/yago-3` (last accessed 2025-09-10).

# 2. Related Work

With SPARQL being a popular query language, there are many publicly available SPARQL engines. A few popular engines are Virtuoso [9], Apache Jena [3], and GraphDB [13]. While Virtuoso started as a relational database management system mainly used with SQL queries, they developed SPARQL support [9] with the growth of the semantic web. Virtuoso also supports text scans inside the SPARQL queries with the predicate *bif:contains*. Similar to QLever, the object then specifies what words to scan for. This scan happens on an inverted full-text index that was built on parts of the ontology. In a similar manner to Virtuoso, Apache Jena provides a full-text index using Lucene [8], which also works as an inverted index. “The basic Jena text extension associates a triple with a document” [2]. Only triples having a literal object are associated with a document. Furthermore, each document also tracks the predicate of the triple. When queried, the full-text index does not return the text containing the word but instead the subject of the literal containing said word. The index can also be queried using a word and a predicate. A query will then return the subject of triples that contain the queried predicate and have an object literal containing the queried word. While the *Text Index Literal Filtering* presented in this work works similarly, filtering triples by their predicate, this is done during index building and not during retrieval. Also like Broccoli [4], QLever differs in another way from these engines. That is, while Virtuoso and Apache Jena provide a full-text index, this index is only built upon the ontology. Broccoli and QLever provide the option to add a separate text corpus that has references to entities of the ontology. While “Broccoli has no query planner and a simplistic KB index” [5], QLever is close to SPARQL 1.1 support. This positions QLever as a state-of-the-art SPARQL engine for large knowledge graphs, uniquely combining efficient SPARQL query processing with deeply integrated full-text search.

# 3. Background

## 3.1 Full-text Index Structure

To understand how QLever used to build, save, and scan its full-text index, we will first look at what an index is used for. The goal of an index is to quickly find and return the correct information for a given query. The term "correct" can mean different things depending on the scenario. In the case of QLever's full-text index, the query can either be a *WordScan* which works on a word or prefix, or an *EntityScan*, which works on RDF entities, or a variable. The *EntityScan* only works together with a *WordScan* since the goal is to find text records where certain words or prefixes co-occur with certain entities. Focusing on the *WordScan*, the information that should be returned is the text in which a word or prefix occurs. QLever provides more information, also returning the score for a given word or prefix. If the query was a prefix, QLever Additionally returns the matching word found in the text. The scores should provide a measure of how important a result is for a given query. One index structure that is often used to index text datasets is a so-called inverted index. An inverted index provides a map that, for a given word, returns all documents in which the word occurs in. In most cases the inverted index provides additional indexed information together with such a result. In this index structure, prefix search cannot easily be executed. A solution for this is a so-called half-inverted index proposed in [7]. Instead of mapping a single word to all its information, the half-inverted index consists of multiple postings for each word. As explained in the introduction, these postings are put into blocks. Inside the blocks, the postings are sorted by text record, while the blocks themselves are ordered by `WordVocabIndex`. The ordering means block $i$ contains smaller `WordVocabIndex`es than block $i + 1$. Since the text dataset that is used to build this index is sorted by text records, the term half-inverted is used. An example of the block structure can be seen in 3.1. The first step for this to work is to assign each word a `WordVocabIndex` in lexicographical order. This means for two `WordVocabIndex`es $i, j \in \mathbb{N}$:

$$i \overset{num}{<} j \Rightarrow Vocab[i] \overset{lex}{<} Vocab[j]$$

Where $Vocab[i], Vocab[j]$ are the words that got assigned the `WordVocabIndex`es $i$ and $j$. Each block knows its first and last `WordVocabIndex`. With this information, the correct block containing the entries the user is looking for can be found in logarithmic time using binary search.

Text Index Block Structure Example:

**Block 1:**

| Text Record Index | Word or Entity Index | Score | Word or Entity |
|---|---|---|---|
| 1 | 0 | 1 | `astronomer` |
| 1 | 1 | 1 | `astronomy` |
| 2 | 0 | 0 | `astronomer` |
| 2 | 1 | 0 | `astronomy` |
| 1 | 0 | 0 | `<Astronomer>` |
| 2 | 0 | 0 | `<Astronomer>` |
| 1 | 1 | 0 | `<Space>` |
| 2 | 1 | 0 | `<Space>` |

**Block 2:**

| Text Record Index | Word or Entity Index | Score | Word or Entity |
|---|---|---|---|
| 1 | 2 | 1 | `space` |
| 1 | 2 | 1 | `space` |
| 1 | 0 | 0 | `<Astronomer>` |
| 2 | 0 | 0 | `<Astronomer>` |
| 1 | 1 | 0 | `<Space>` |
| 2 | 1 | 0 | `<Space>` |

Figure 3.1: An example of how blocks look like in the text index. Each block has a list of words and a separate list of entities. Each entity co-occurs in a text record with at least one of the words of the word list. The entity lists are deduplicated. This can be seen in Block 1. The word "astronomer" appears together with the word "astronomy" and the entity "<Astronomer>" in text record 1. Instead of saving "<Astronomer>" twice with the same `TextRecordIndex`, it is only saved once. The same thing happens again for text record 2. Note the written words and entities right of the bar are only to make the blocks readable. The first block contains Word Indices in the range of [0, 1] while the second block contains Word Indices in the range of [2, 2]. Therefore the first block comes before the second in the actual index.

## 3.2 Text Corpus

Now we will look at the dataset the full-text index is built upon. We will first look at the `wordsfile` and `docsfile` as they provide an understanding of what type of information the full-text index building takes as input. An excerpt of the actual scientists `docsfile` can be seen in Figure 3.2.

Docsfile excerpt:

| DocumentIndex | Document |
|---|---|
| 4 | An astronomer is a scientist in the field of astronomy who concentrates their studies on a specific question or field outside of the scope of Earth. |
| 7 | They look at stars, planets, moons, comets and galaxies, as well as many other celestial objects — either in observational astronomy, in analyzing the data or in theoretical astronomy. |
| 22 | Examples of topics or fields astronomers work on include: planetary science, solar astronomy, the origin or evolution of stars, or the formation of galaxies. |
| 25 | There are also related but distinct subjects like cosmology which studies the Universe as a whole. |

Figure 3.2: Excerpt of the scientists `docsfile`.

The `docsfile` has a column for a `DocumentIndex` and a column for the document. This file gets compressed to the 'docsDB' which is later used to retrieve the texts for certain `TextRecordIndex`es. To annotate entities to documents, each document was split up into smaller text records. The `wordsfile` then contains all words of the text records except the stop words. How the first document from Figure 3.2 was split up can be seen in Figure 3.3. In the example seen in Figure 3.3 one document was split into four text records. The relation between `DocumentIndex` and `TextRecordIndex` is the following:

For `DocumentIndex`es: $i, j \in \mathbb{N}$ with:

$$i < j, \nexists k \in \mathbb{N} \text{ with } i < k < j$$

All `TextRecordIndex`es: $l \in \mathbb{N}$ with:

$$i < l \leq j \text{ belong to } \texttt{DocumentIndex}\ j$$

Besides words, the `wordsfile` also contains entity mentions like `<Astronomer>` which were assigned to a word of the text record. An excerpt of the `wordsfile` from the scientists dataset can be seen in Figure 3.4. The first column of the `wordsfile` is used for words as well as entities. The second column specifies whether the entry in the first column is a word or an entity. The third column links the word or entity to its source text

Document to text records:

| **Document:** | **Text Records:** |
|---|---|
| An astronomer is a scientist in the field of astronomy who concentrates their studies on a specific question or field outside of the scope of Earth. | An astronomer is a scientist in the field of astronomy |
| | An astronomer is a scientist in the field of astronomy |
| | astronomy who concentrates their studies on a specific question or field outside the scope of Earth. |
| | astronomy who concentrates their studies on a specific question or field outside the scope of Earth. |

Figure 3.3: An example how a single document is converted into multiple overlapping text records. This example is from the scientists `wordsfile` and `docsfile`.

record with a `TextRecordIndex`. The fourth column contains pre-calculated scores.

Wordsfile excerpt:

| Word or Entity | isEntity | TextRecordIndex | Score |
|---|---|---|---|
| `astronomer` | 0 | 1 | 1 |
| `<Astronomer>` | 1 | 1 | 0 |
| `scientist` | 0 | 1 | 1 |
| `field` | 0 | 1 | 1 |
| `astronomy` | 0 | 1 | 1 |
| `astronomer` | 0 | 2 | 0 |
| `<Astronomer>` | 1 | 2 | 0 |
| `:s:firstsentence` | 0 | 2 | 0 |
| `scientist` | 0 | 2 | 0 |
| `field` | 0 | 2 | 0 |
| `astronomy` | 0 | 2 | 0 |
| `astronomy` | 0 | 3 | 1 |
| `concentrates` | 0 | 3 | 1 |
| `studies` | 0 | 3 | 1 |
| `specific` | 0 | 3 | 1 |
| `question` | 0 | 3 | 1 |
| `outside` | 0 | 3 | 1 |
| `scope` | 0 | 3 | 1 |
| `earth` | 0 | 3 | 1 |
| `astronomy` | 0 | 4 | 1 |
| `concentrates` | 0 | 4 | 1 |
| `studies` | 0 | 4 | 1 |
| `field` | 0 | 4 | 1 |
| `outside` | 0 | 4 | 1 |
| `scope` | 0 | 4 | 1 |
| `earth` | 0 | 4 | 1 |

Figure 3.4: Excerpt of the scientists `wordsfile`.

## 3.3 Full-text Index Building

The `wordsfile` and `docsfile` are not necessary to build the full-text index since the literals of the respective RDF database can be used. Both options can also be used together. In the following, we will look at the full-text index building with both options used together.

#### Load the RDF Vocabulary

First the RDF vocabulary is loaded from disk.

### Create the Text Vocabulary

To create the text vocabulary, the `wordsfile` and the RDF vocabulary are parsed. All words from the `wordsfile` are collected in a set. For the RDF vocabulary each literal is split into words using a tokenizer. The tokenizer splits strings at all non-alphanumerical characters. Those words are then collected in the same set. This set is first sorted and then parsed, assigning each word a unique `WordVocabIndex`. This leads to the `WordVocabIndex`es being assigned in lexicographical order.

### Get the Scoring Data

If the scoring metric was specified to be BM25 or TF-IDF, then information for these two metrics is gathered. To gather these two metrics, the `docsfile` is parsed once. During the parsing, an inverted index is built, collecting information on how often and in which documents words occur, as well as how long the documents are. During the building of the half-inverted text index table, these values are used to calculate the respective score.

### Calculate the Block Boundaries

In this step, the block boundaries for the half-inverted text index are calculated. For this, the lexicographically ordered text vocabulary is parsed. Block boundaries are set whenever a word is encountered that either: Has a next word with a new four-letter prefix, has a next word shorter than four letters, is shorter than four letters, or is the last word of the vocabulary. This leads to each word shorter than four letters having an own block, and all other words being in their respective four-letter prefix blocks. Note that the blocks have not been written yet.

### Build the half-inverted Text Index Table

In the next step, the words and entities are processed for the half-inverted text index. To collect all information, a table is used. This table has five columns, as seen in Figure 3.5. The first column is for the `BlockIndex`es that govern which block the entry is written to later. The second column is a flag for whether the entry is a word or an entity. The third column contains the `TextRecordIndex`es. The fourth column contains the `Word-VocabIndex`es for words and the `VocabIndex`es for entities. The last column contains the `Score`s. Once again the `wordsfile` and RDF vocabulary are parsed. The words in the `wordsfile` are pre-sorted by `TextRecordIndex`. Due to this ordering, we can collect all information of a text record in two maps. One map for words and their scores and one map for entities and their scores. If scoring is not set to BM25 and TF-IDF, the scores present in the `wordsfile` are used. If the scores of the `wordsfile` are used and words or entities occur multiple times in one text record, the scores are summed up. When the next text record is reached, the two maps are used to add the whole previous text record to the table. For each word in the word map, the `BlockIndex` is looked up, and all necessary information is written as a row to the table. During this, all distinct `BlockIndex`es touched by the words are tracked. All entities of the text record are added once for each distinct `BlockIndex` together with their score. This duplication is necessary since later only one block is returned, and the block needs to contain all entities co-occurring with any word of the block. This collecting of text records and then writing them to the table is first done for the `wordsfile` and afterward for the RDF literals. The literals are treated as text records containing one entity: the literal itself. An example table that is filled using the first four text records of the scientists text corpus can be seen in Figure 3.5.

#### Sort the half-inverted Text Index Table

After collecting all the information, the table is sorted by the order of the columns. `BlockIndex` first, then by the flag whether the entry is a word or entity, then by the `TextRecordIndex`, then by the `WordVocabIndex` or the `VocabIndex`, and lastly by `Score`.

#### Write the half-inverted Text Index to disk

After sorting, the half-inverted text index table is parsed. This time, block by block instead of text record by text record. For each block, all word postings and all entity postings are collected. Due to the sorting of the table, this leads to two lists. One list for the words and one for the entities. Both lists are sorted by `TextRecordIndex`, then by `WordVocabIndex` or `VocabIndex`, and lastly by `Score`. This can be seen in Figure 3.1. The word list is compressed and saved to file column by column, first `TextRecordIndex`es, then `WordVocabIndex`es, and then `Score`s. After the word postings are written to disk, the entity postings are written in the same manner. At the end of each block, its metadata is written, which contains information like the first and last `WordVocabIndex`. After writing all blocks, metadata for the whole full-text index is written.

#### Build the 'docsDB'

As a last step, the `docsfile` is parsed and written to a file. The file keeps track of where a certain document starts. This file can later be used to populate a map that, for a given `TextRecordIndex`, returns the matching document.

Old half-inverted text index table example:

| Block Index | Text Record Index | isEntity | Word or Entity Index | Score | Word or Entity |
|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | `:s:firstsentence` |
| 1 | 1 | 0 | 1 | 1 | `astronomer` |
| 1 | 1 | 0 | 2 | 1 | `astronomy` |
| 1 | 2 | 0 | 1 | 0 | `astronomer` |
| 1 | 2 | 0 | 2 | 0 | `astronomy` |
| 1 | 3 | 0 | 2 | 1 | `astronomy` |
| 1 | 4 | 0 | 2 | 1 | `astronomy` |
| 1 | 1 | 1 | 0 | 0 | `<Astronomer>` |
| 1 | 2 | 1 | 0 | 0 | `<Astronomer>` |
| 2 | 3 | 0 | 3 | 1 | `concentrates` |
| 2 | 4 | 0 | 3 | 1 | `concentrates` |
| 3 | 3 | 0 | 4 | 1 | `earth` |
| 3 | 4 | 0 | 4 | 1 | `earth` |
| 4 | 1 | 0 | 5 | 1 | `field` |
| 4 | 2 | 0 | 5 | 0 | `field` |
| 4 | 4 | 0 | 5 | 1 | `field` |
| 4 | 1 | 1 | 0 | 0 | `<Astronomer>` |
| 4 | 2 | 1 | 0 | 0 | `<Astronomer>` |
| 5 | 3 | 0 | 6 | 1 | `outside` |
| 5 | 4 | 0 | 6 | 1 | `outside` |
| 6 | 3 | 0 | 7 | 1 | `question` |
| 7 | 1 | 0 | 8 | 1 | `scientist` |
| 7 | 2 | 0 | 8 | 0 | `scientist` |
| 7 | 1 | 1 | 0 | 0 | `<Astronomer>` |
| 7 | 2 | 1 | 0 | 0 | `<Astronomer>` |
| 8 | 3 | 0 | 9 | 1 | `scope` |
| 8 | 4 | 0 | 9 | 1 | `scope` |
| 9 | 3 | 0 | 10 | 1 | `specific` |
| 10 | 3 | 0 | 11 | 1 | `studies` |
| 10 | 4 | 0 | 11 | 1 | `studies` |

Figure 3.5: Old half-inverted text index table example from the scientists text corpus. Note this example shows a sorted table.

## 3.4 Full-text Index Retrieval

During the retrieval, the user has two main options to scan the full-text index. They can either use a *WordScan* or an *EntityScan*. The *WordScan* is executed with the keyword *ql:contains-word*. This keyword can be used as a predicate in a SPARQL query. The respective object is a string of one or multiple words or prefixes, and the subject is the variable used to match text records. A simple *WordScan* and the result of it can be seen in Figure 3.6 and Figure 3.7.

Example *WordScan* query:

```
SELECT * WHERE {
    ?doc ql:contains-word "test" .
}
```

Figure 3.6: Example query for a simple *WordScan*.

Results excerpt:

| ?ql_score_word_doc_test | ?doc |
|---|---|
| 1 | The theory of mind hypothesis is supported by the atypical responses . . . |
| 1 | Commercial availability of tests may precede adequate understanding . . . |
| 1 | In Aristotelian science, especially in biology, things he saw himself have . . . |
| 1 | The new 1901 Constitution of Alabama included provisions for voter . . . |

Figure 3.7: Excerpt of the results of the query seen in Figure 3.6 executed on the Freebase easy database. Query was executed on 2025-08-13 at 11:31:15 on the QLever website. Link to query: `https://qlever.cs.uni-freiburg.de/fbeasy/H7UGBm`

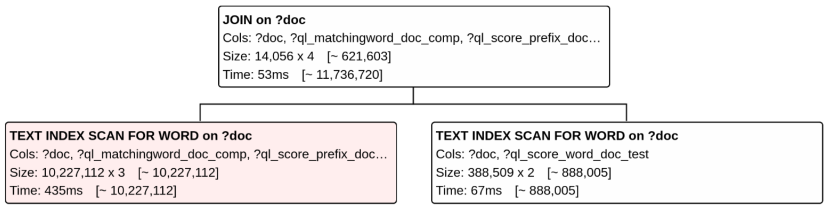
To understand later changes, we will look at the *WordScan* internally. First, the query is parsed. For the query seen in Figure 3.6, the query tree only consists of one single operation. That operation is the *WordScan*. The *WordScan* looks up the `WordVocab-Index` range for the queried word or prefix. For a word, this range only consists of one `WordVocabIndex`, but for a prefix, the range spans multiple `WordVocabIndex`es. With the larger value of this range, a binary search is performed to find the block containing the word or prefix. Since the search only allows prefixes of a minimum length of four, it is checked whether the smaller `WordVocabIndex` is in the selected block. If this is not the case, the query fails, since only one block can be returned. The word list of the selected block is then decompressed into a table with the columns `TextRecordIndex`, `WordVocabIndex`, and `Score`. If the block contains words outside the queried range, the table is parsed and the unnecessary words are filtered. If the *WordScan* was done using a prefix, the column for the `WordVocabIndex`es is kept; else, it is deleted. To show the user a human- readable result, the `TextRecordIndex`es are replaced by the related documents from the 'docsDB'. Note multiple different `TextRecordIndex`es can point to a single document due to the relation described in Figure 3.3. If the column containing the `WordVocabIndex`es was not removed, the `WordVocabIndex`es are replaced by the respective words. A query with multiple words or prefixes results in multiple *WordScan*s that are then joined on the `TextRecordIndex` column. Such a query and the resulting query execution tree can be seen in Figure 3.8 and Figure 3.9.

Example multiple *WordScan* query:

```
SELECT * WHERE {
    ?doc ql:contains-word "test␣comp*" .
}
```

Figure 3.8: Example query where object of *ql:contains-word* has multiple words.

Multiple *WordScan* query execution tree:



```
JOIN on ?doc
Cols: ?doc, ?ql_matchingword_doc_comp, ?ql_score_prefix_doc…
Size: 14,056 x 4   [~ 621,603]
Time: 53ms   [~ 11,736,720]

TEXT INDEX SCAN FOR WORD on ?doc
Cols: ?doc, ?ql_matchingword_doc_comp, ?ql_score_prefix_doc…
Size: 10,227,112 x 3   [~ 10,227,112]
Time: 435ms   [~ 10,227,112]

TEXT INDEX SCAN FOR WORD on ?doc
Cols: ?doc, ?ql_score_word_doc_test
Size: 388,509 x 2   [~ 888,005]
Time: 67ms   [~ 888,005]
```

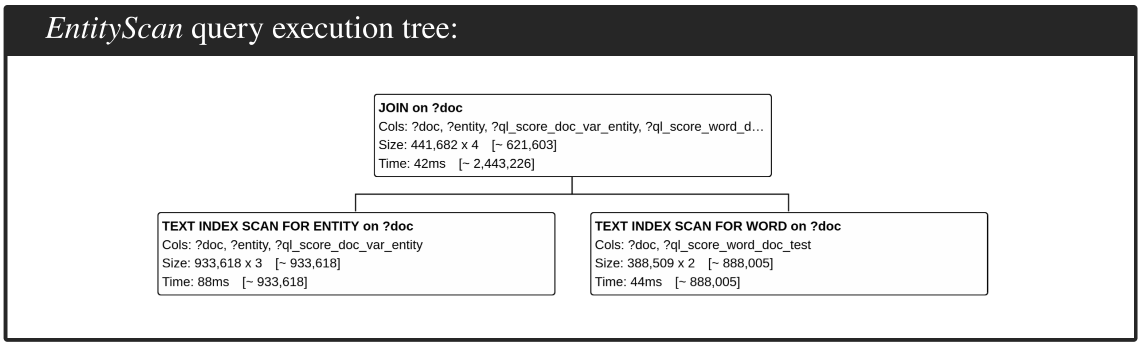
Figure 3.9: Query execution tree of Figure 3.8. Query was executed on the Freebase easy dataset on 2025-08-13 at 17:17:07 using the QLever website. Link to query: `https://qlever.cs.uni-freiburg.de/fbeasy/2DmLm7`

The *EntityScan* works similarly to the *WordScan*. As explained above, the query cannot execute an *EntityScan* without a *WordScan* on the same text record variable. This is because the *EntityScan* is used to find entities co-occurring with words in certain text records. In the query planning step for each *ql:contains-entity*, it is looked up what *WordScan*s are planned on the same text record variable. For each *WordScan*, all words and prefixes found are collected. Then it is looked up which word or prefix has the smallest entity list in its block. The word or prefix with the smallest entity list in its block is then assigned to the *EntityScan*. Afterward, the process to find the correct block is the same as for the *WordScan*. If the block containing the assigned word or prefix is found, the entity list of the block is decompressed into a table. The table is the same as for the *WordScan*, but instead of the second column containing `WordVocabIndex`es, the column contains `VocabIndex`es that correspond to entities of the RDF vocabulary. *EntityScan*s can be done using a variable entity or a fixed entity. This means we can either scan for all entities co-occurring with a word or look for a specific entity. If only scanning for a fixed entity, the table is filtered for said entity. The intermediate result table now contains all variable or fixed entity postings for all `TextRecordIndex`es present in the block. Since not all `TextRecordIndex`es present in this block belong to text records containing the query word or prefix, the *EntityScan* is joined with a *WordScan* on the `TextRecordIndex` variable. This leaves only the text records where the queried words or prefixes co-occur with the entities. One simple example query and the respective query execution tree can be seen in Figure 3.10 and Figure 3.11.

Example *EntityScan* query:

```
SELECT * WHERE {
    ?doc ql:contains-word "test" .
    ?doc ql:contains-entity ?entity .
}
```

Figure 3.10: Example query for a simple *EntityScan*.


*EntityScan* query execution tree:

JOIN on ?doc
Cols: ?doc, ?entity, ?ql_score_doc_var_entity, ?ql_score_word_d…
Size: 441,682 x 4 [~ 621,603]
Time: 42ms [~ 2,443,226]

TEXT INDEX SCAN FOR ENTITY on ?doc
Cols: ?doc, ?entity, ?ql_score_doc_var_entity
Size: 933,618 x 3 [~ 933,618]
Time: 88ms [~ 933,618]

TEXT INDEX SCAN FOR WORD on ?doc
Cols: ?doc, ?ql_score_word_doc_test
Size: 388,509 x 2 [~ 888,005]
Time: 44ms [~ 888,005]

Figure 3.11: Query execution tree of 3.10. Query was executed on the Freebase easy dataset on 2025-08-13 at 17:57:20 using the QLever website. Link to query: `https://qlever.cs.uni-freiburg.de/fbeasy/eSFLEp`

## 3.5 Theoretical Analysis

In this section we will look at the theoretical runtime of the full-text index building as well as the runtime of the *WordScan* and the *EntityScan*.

### 3.5.1 Full-text Index Building

As seen above, the steps for full-text index building are the following:

1. Load the RDF Vocabulary
2. Create the Text Vocabulary
3. Get the Scoring Data
4. Calculate the Block Boundaries
5. Build the half-inverted Text Index Table
6. Sort the half-inverted Text Index Table
7. Write the half-inverted Text Index to disk
8. Build the 'docsDB'

Since the steps are executed in order we will calculate the runtime of each step and at the end see which term asymptotically outgrows the others.

### Loading the RDF Vocabulary

Let $t_1$ be the runtime of loading the RDF vocabulary from disk. The operation depends on the size of the RDF vocabulary. This leads to:

$$\begin{aligned} t_1 &\in O(R) \\ R &= \text{size of the RDF vocabulary} \end{aligned}$$

### Create the Text Vocabulary

Let $t_2$ be the runtime of building the text vocabulary. To build the text vocabulary the whole `wordsfile` and optionally the RDF vocabulary is parsed. For each line in the `wordsfile` that is marked as word line and not entity line, the word is added to a hash set. Afterward, the RDF vocabulary is parsed. Each literal is split up into single words and each word is added to the hash set. At a last step the hash set is sorted and parsed, to assign each word a `WordVocabIndex` in lexicographical order. To calculate $t_2$ we will split it up into sections:

$t_2^1$ = Time to parse the `wordsfile` and add the words to the hash set
$t_2^2$ = Time to parse the RDF vocabulary and add the words of the literals to the hash set
$t_2^3$ = Time to build the text vocabulary from the hash set

For the parsing the `wordsfile` the size of the `wordsfile` is relevant. For each word a hash set operation is performed and for each entity the line is skipped. This leads to:

$$\begin{aligned} t_2^1 &\in O(W) \\ W &= \text{\# lines in the } \texttt{wordsfile} \end{aligned}$$

For the parsing of the RDF vocabulary each non literal entity is skipped. Each literal is split into single words and for each word a hash set operation is performed. This leads to:

$$\begin{aligned} t_2^2 &\in O(R_{NL} + L) \\ R_{NL} &= \text{\# non-literals in the RDF vocabulary} \\ L &= \text{total \# words in all literals} \\ R &= \text{size of the RDF vocabulary} \end{aligned}$$

The relation between $R_{NL}$, $R$, and $L$ is complicated. With the assumption that $R_{NL} \propto R \propto L$ which is often the case in the real application this can be simplified to:

$$t_2^2 \in O(R)$$

To build the vocabulary the hash set with all words is sorted and then each word gets assigned a `WordVocabIndex`. The runtime of the sorting asymptotically outgrows the runtime of assigning the `WordVocabIndex`es. This leads to:

$$\begin{aligned} t_2^3 &\in O(V \cdot log(V)) \\ V &= \#\text{ distinct words of the } \texttt{wordsfile} \text{ and of all literals} \\ &= \text{Size of the text vocabulary} \end{aligned}$$

Heap's Law [10] says that the size of a vocabulary $V$ is proportional to $M^{\lambda}$ where $M$ is the number of words the vocabulary is built upon and $0 < \lambda < 1$. Together with the assumption that $L \propto R$ this leads to:

$$\begin{aligned} & V \propto (W + L)^{\lambda} \propto (W + R)^{\lambda} \\ \Rightarrow\ & t_2^3 \in O((W + R)^{\lambda} \cdot log((W + R)^{\lambda})) \\ \Rightarrow\ & t_2^3 \in O((W + R)^{\lambda} \cdot \lambda \cdot log((W + R))) \\ \Rightarrow\ & t_2^3 \in O((W + R)^{\lambda} \cdot log(W + R)) \end{aligned}$$

Out of the three times, $t_2^3$ asymptotically outgrows $t_2^1$ and $t_2^2$. This leads to:

$$t_2 \in O((W + R)^{\lambda} \cdot log(W + R))$$

### Get the Scoring Data

Let $t_3$ be the runtime of getting the scoring data. To get the scoring data an inverted index is built. This inverted index maps `WordVocabIndex`es to an inner map, that then maps `DocumentIndex`es to `TermFrequencies`. Together with this main map another map is used to keep track of the length of documents. To build the inverted index and the document length map, all documents are parsed. For each word in a document it is looked up whether the word exists in the text vocabulary. If it exists, the inner map for the `WordVocabIndex` is retrieved, and the `TermFrequency` for the respective document is increased. Afterward, the entry in the document length map for the `DocumentIndex` is increased by one. In terms of runtime this means: For each word, that is encountered in a document, and that is present in the text vocabulary, there are three hash operations. Each word that is not in the text vocabulary is skipped. Since the number of total words in all documents is proportional to the number of lines of the `wordsfile` this leads to:

$$t_3 \in O(W)$$

#### Calculate the Block Boundaries

Let $t_4$ be the runtime of calculating the block boundaries. To calculate the block boundaries the whole text vocabulary is parsed and at each new word that is shorter than four letters or that starts with a new four letter prefix a block boundary is set. The action of setting a block boundary works by saving the last `WordVocabIndex` of a block. This leads to:

$$\begin{aligned} t_4 &\in O(V) \\ \Rightarrow t_4 &\in O((W+R)^{\lambda}) \end{aligned}$$

#### Build the half-inverted Text Index Table

Let $t_5$ be the runtime of building the half-inverted text index table. To build the table, all text records are collected and written to the table. Each entity of a text record is written once for each block touched by the words of a text record. Under the realistic assumption that no text record contains words of all blocks, the number of touched blocks is proportional to the number of words in the text record. Since entities were assigned to words of the text records, this means the number of entities in a text record is also proportional to the number of words in a text record. Let $T_{max}$ be the number of words in the largest text. This means we have a maximum of $T_{max}^2$ number of writing operations for each text record. Since the number of text records is proportional to $(W + R)$, this leads to:

$$t_5 \in O((W+R) \cdot (T_{max})^2)$$

The assumption that the maximum size of a text record does not grow with the total number of text records after a certain threshold is reached, leads to $T_{max}$ being a constant. This results in:

$$t_5 \in O(W+R)$$

#### Sort the half-inverted Text Index Table

Let $t_6$ be the runtime of sorting the half-inverted index table. To calculate the runtime of this we need to evaluate the size of the half-inverted index table $I$. In calculating the runtime to build this table we saw it can be built in $O(W + R)$. From this $I \propto (W + R)$ follows, which leads to:

$$t_6 \in O((W+R) \cdot log(W+R))$$

### Write the half-inverted Index to disk

Let $t_7$ be the runtime of writing the half-inverted index table. Each write operation has a constant cost in terms of the half-inverted index table size. This leads to:

$$t_7 \in O(W + R)$$

### Build the 'docsDB'

To build the 'docsDB' the `docsfile` is parsed and each entry is written to a file while keeping track of the end position of each document. Let $t_8$ be the time building the 'docsDB' takes. Then:

$$\begin{aligned} t_8 &\in O(D) \\ \Rightarrow t_8 &\in O(W) \end{aligned}$$

### Overall Runtime of the full-text Index Building

Let $t$ be the overall runtime. To calculate the asymptotic growth of $t$ we will look at the growth rates of $t_1$ to $t_8$:

$$\begin{aligned} t_1 &\in O(R) \\ t_2 &\in O((W + R)^\lambda \cdot log(W + R)) \\ t_3 &\in O(W) \\ t_4 &\in O((W + R)^\lambda) \\ t_5 &\in O(W + R) \\ t_6 &\in O((W + R) \cdot log(W + R)) \\ t_7 &\in O(W + R) \\ t_8 &\in O(W) \end{aligned}$$

We can immediately see when comparing asymptotic growth:

$$t_1, t_3, t_8 < t_5, t_7 < t_6$$

Since $0 < \lambda < 1$ it also holds that:

$$\begin{aligned} t_2 &< t_6 \\ t_4 &< t_5, t_7 < t_6 \end{aligned}$$

Therefore $t_6$ dominates asymptotically leading to:

$$t \in O((W + R) \cdot log(W + R))$$

Under the assumptions that the number of total words in all literals is proportional to the RDF vocabulary size. And that the number of maximum words in a single text record is not proportional to the number of total text records after a certain threshold.

### 3.5.2 *WordScan*

The *WordScan* has four different steps:

1. Look up the correct block for the given `WordVocabIndex` range
2. Read word list of the block
3. Filter the intermediate result
4. Replace IDs by readable values

We assume the *WordScan* is the only operation in the query. In this case we include the last step replacing the IDs by readable values since QLever does this as last step for every query.

#### Find the correct block

Finding the correct block for a given `WordVocabIndex` range is rather simple. During full-text index building the largest `WordVocabIndex` of each block has been written to the metadata in ascending order. Therefore, we can use the upper value of the `Word-VocabIndex` range to perform a binary search on this list.

Let $t_1$ be the runtime of finding the correct block. Let $B$ be the number of blocks written during full-text index building. Then:

$$t_1 \in O(log(B))$$

#### Read the word list of the block

Let $t_2$ be the runtime of reading the word list of the block. Let $B_i^w$ be the size of the word list of block $i$ that should be returned. Then:

$$t_2 \in O(B_i^w)$$

#### Filter the intermediate result

Let $t_3$ be the runtime of filtering the intermediate result. Then:

$$t_3 \in O(B_i^w)$$

#### Replace the IDs by readable values

Let $t_4$ be the runtime of replacing the IDs by readable values. Let $S$ be the size of the table. We know $S \propto B_i^w$. Then:

$$t_4 \in O(B_i^w)$$

#### Overall Runtime for a *WordScan*

Let $t$ be the runtime of a *WordScan*. Then:

$$t \in O(log(B) + B_i^w)$$

### 3.5.3 *EntityScan*

Since the *EntityScan* finds all entities co-occurring with a word a prefix it has an assigned word or prefix. Using this, the *EntityScan* works similar to the *WordScan* except the filtering step. Instead of filtering, the intermediate result is joined with a *WordScan*. For simplicity we will assume only one *WordScan* is used together with an *EntityScan*. The steps then are:

1. Look up the correct block for the given `WordVocabIndex` range
2. Read entity list of the block
3. Join the two sorted results
4. Replacing IDs by readable values

#### Find the correct block

This is exactly the same as for the *WordScan*, therefore:

$$t_1 \in O(log(B))$$

#### Read the entity list of the block

Let $B_i^e$ be the size of the entity list of block $i$ that should be returned. Then:

$$t_2 \in O(B_i^e)$$

### Join the two sorted results

Let $t_3$ be the runtime of joining the two sorted results. Let $S$ be the output size of this operation. The output size depends on the type of *WordScan* used. If the *WordScan* was executed on a prefix this can lead to the same `TextRecordIndex` appearing multiple times in the *WordScan* result. When joining the *EntityScan* result with the *WordScan* result, each row of the *EntityScan* containing a certain `TextRecordIndex` is combined with each row of the *WordScan* having the same `TextRecordIndex`. If the *EntityScan* was executed together with a single prefix scan this leads to a worst case of:

$$S = B_i^e \cdot B_i^w$$
$$\Rightarrow t_3 \in O(B_i^e \cdot B_i^w)$$

Since the size of the word list of a block is proportional to the size of the entity list this leads to:

$$\Rightarrow t_3 \in O((B_i^e)^2)$$

### Replace the IDs by readable values

Let $t_4$ be the runtime of replacing the IDs by readable values. Let $S$ be the size of the table. The same upper bound hold as above. This leads to:

$$t_4 \in O((B_i^e)^2)$$

### Overall Runtime for an *EntityScan*

Let $t$ be the runtime of an *EntityScan*. Then:

$$t \in O(log(B) + (B_i^e)^2)$$

# 4. Text Index Literal Filtering

As explained in the introduction, the full-text index can be built or extended using literals of the RDF vocabulary. We saw the literals could either be strings or a mix of a value and an IRI specifying the datatype of the value. In QLever, the literals that are a mix of a datatype and an IRI are classified if possible. One example IRI commonly used in literals is `<http://www.w3.org/2001/XMLSchema#int>`. Internally, literals with this IRI are classified as integers and not as literals. But there are many dataset-specific IRIs used in literals, which leads to QLever classifying these mixed literals as literals. As a result, many literals are irrelevant for the full-text index since they, for example, only contain a number or a label. Another problem of the old building of the full-text index from literals was the missing retrieval of texts, as well as the overhead of parsing the whole RDF vocabulary instead of only the literals.

## 4.1 Feature changes

All three issues mentioned above can be solved with the filtering of triples and pre-saving of literals during the RDF index building. To filter the most important literals, we can look at certain predicates. For example, the predicate "<description>" is probably followed by a more informative literal in terms of the full-text index than the predicate "<label>". A good query to get an understanding of the important predicates in a dataset can be seen in Figure 4.1.

SPARQL query to find relevant predicates:

```
SELECT ?p (COUNT(?o) AS ?count)
(SAMPLE(?o) AS ?exampleO) WHERE {
    ?s ?p ?o .
}
GROUP BY ?p
ORDER BY DESC(?count)
```

Figure 4.1: This query can be used to find possible predicates that hint on qualitative object literals for the full-text index.

A regex filter was implemented that can be used during RDF index building to filter triples by their predicate. The regex filter looks for a partial match. A full match syntax can still be achieved using the standard regex syntax "^fullMatch$". The user can specify whether the filter should work as a whitelist or a blacklist. If the filter works as a whitelist, all literal objects of matching triples are added to the full-text index. If the filter works as a blacklist, all literal objects of non-matching triples are added to the full-text index. The filtering step is performed during the input parsing phase of the RDF index building.

After the filtering is performed, all literals marked as 'inTextIndex' are written to a file as their `VocabIndex`es. This file is called the literal file. If the same literal appears with different values for 'inTextIndex', it is added to the literal file as long as there is one occurrence where 'inTextIndex' is true. The literal file is used during the full-text index building together with the RDF vocabulary to retrieve the literals using the `VocabIndex`es instead of parsing the whole RDF vocabulary.

If the user builds the full-text index upon an old RDF index that does not have the literal file, QLever reverts to the old parsing of the whole RDF vocabulary. It is also not possible to specify a regex when only adding the full-text index.

The literal file is also used for the retrieval. If a text record index i returned in the result of a *WordScan* or an *EntityScan* is out of range of the `DocumentIndex`es in the 'docsDB', we know it belongs to a literal. Since each literal added to the text index increases the number of text records by one, we can subtract the highest `DocumentIndex` of the 'docsDB' from i and get index j. We then know that the j-th literal of the literal file is the requested text record. This solves the problem of empty results when literals should be returned.

## 4.2 Usage

The *Text Index Literal Filtering* adds two options to the index builder of QLever and extends the meaning of one already available option:

- –text-index-regex, -r [option]: The option given is the regex used to match predicates.
- –text-index-regex-is-blacklist, -R: If this flag is set the filtering works as blacklist, excluding all literal objects of matching triples and adding all literal objects of non-matching triples.
- –text-words-from-literals, -W: This was the option to add literals to the full-text index. Now this flag works as indicator to add all literals to the full-text index regardless if they appear as objects or subjects. It cannot be used together with the '-r' option.

## 4.3 Theoretical Analysis

We will evaluate the changes made to the runtime in two different parts. The full-text index building and the retrieval.

### 4.3.1 Full-text Index Building

#### Without Filtering

We will first look at the saving of the literals without filtering. This change modifies the runtime of each step where the whole RDF vocabulary was parsed to find the literals. This improves times when building the text vocabulary and the half-inverted text index table, but it does not change the asymptotic runtime of the full-text index building. Meaning the total runtime $t$ remains in $O((W + R) \cdot log(W + R))$.

#### With Filtering

If we look at the way filtering influences full-text index building we see a major improvement. Depending on the filter used, the number of words in all literals that get written to the full-text index decreases. This modifies to proportional relation between $L$ and $R$ from $L = k \cdot R$ to $L = l \cdot R$ with $l < k$. How much smaller $l$ is than $k$ depends on the filter chosen. In theory this can improve runtime in a noticeable way, but does not change the asymptotic runtime.

### 4.3.2 Retrieval

During the retrieval two things change. First, since there are potentially fewer text records and words the block sizes and the number of blocks are reduced. In theory, this can improve retrieval times. Second, the cost to look up the value for the `TextRecord-Index` changes since now `TextRecordIndex`es larger than the 'docsDB' correspond to literals. This cost still remains constant, but is slightly increased. In theory, this can slightly worsen retrieval times.

# 5. Document only Text Index Building

In section 3.2 we looked at the `wordsfile` and `docsfile`. While the `docsfile` represents a common form of a text database, the `wordsfile` is rather complicated. Since QLever can be used as an engine for self-created databases, this format is an obstacle for users to build their own full-text index. The reason for the format of the `wordsfile` is the automatic extraction of entities from documents. In [6] it is explained that documents have to be split up into smaller so-called contexts to extract triple relations from texts. The so-called contexts are referred to as text records in the following. The example shown in [6] is the sentence *"Ruth Gabriel, daughter of the actress and writer Ana Maria Bueno, was born in San Francisco"*. This sentence contains information that needs multiple triples to be described. Therefore, this one sentence is split up into the following text records:

1. *Ruth Gabriel was born in San Francisco*
2. *Ruth Gabriel, daughter of Ana Maria Bueno*
3. *actress Ana Maria Bueno*
4. *writer Ana Maria Bueno*

In this format it is easier to annotate entities to words. While this explains the reason for the format of the `wordsfile`, QLever itself or the QLever wiki does not provide a method for the user to create such a `wordsfile` from a given `docsfile`. Besides the `wordsfile` not being easily reproducible, this leads to other problems as well. First, if a document contains a word, for example, "space" and an entity, for example, "<Astronomer>", but the distance between them is to high, they do not appear in the same text record. This leads to the query seen in Figure 7.15 not finding them using the old full-text index. The second problem that can be seen in Figure 7.16 is the repetition of texts. This stems from the relation between `TextRecordIndex`es and `DocumentIndex`es explained in section 3.2. Multiple `TextRecordIndex`es are matched to single `DocumentIndex`es and therefore return the same text. This provides no extra information for the user and clutters the result.

## 5.1 Feature changes

To combat the above-mentioned problems, QLever now provides two more options for the full-text index building. The first option is to build the full-text index only from the `docsfile`. This leads to the full-text index being completely separated from the main RDF index. While this might be wanted for some special cases, it removes the important feature of a deeply connected SPARQL+Text engine. To still provide the possibility to connect entities to documents, the second new option allows users to only add entities from a `wordsfile`. In both cases, pre-computed scores for words are lost.

Building the full-text index from only the `docsfile` does not change much during full-text index building. Instead of parsing the `wordsfile` to build the vocabulary or the half-inverted index, the `docsfile` is parsed. Each document is then split up the same way as the literals, and each word is transformed to the line format of the `wordsfile`. For example, the word "puddle" from a document with `DocumentIndex` 5 is transformed to:

| Word or Entity | isEntity | TextRecordIndex | Score |
|---|---|---|---|
| `puddle` | 0 | 5 | 0 |

If the option to only add entities from the `wordsfile` is used, then the `wordsfile` and `docsfile` are parsed in parallel during the half-inverted index building. This needs to be done since all word and entity mentions for a text record have to be collected before the next text record is parsed.

## 5.2 Usage

The *Document only Text Index Building* feature adds two options to the QLever index builder:

- –text-words-from-docsfile, -D: If this flag is given the `docsfile` is parsed for the full-text index instead of the `wordsfile`.
- –text-entities-from-wordsfile, -E: This flag can only be used together with the '-D' option. If this flag is given the `wordsfile` is used to link entities to documents.

## 5.3 Theoretical Analysis

We will look at the runtime of the full-text index building using a `wordsfile` that only contains entities and using the `docsfile` for the words.

### 5.3.1 Full-text Index Building

Choosing to build from the `docsfile` essentially means transforming every word in the `docsfile` to a word of the `wordsfile`. This means we can use the old runtime analysis but with a changed proportion between the number of documents, $D$, and the number of lines in the `wordsfile`, $W$. Whereas before the relation was $W = k \cdot D$, it now is modified to $W = l \cdot D$. It is not easy to say whether $k$ or $l$ is bigger since the `wordsfile` did not contain stop words from the documents but at the same time repeated the non-filtered words due to the splitting into overlapping text records. For asymptotic growth, this does not change anything. Therefore, the total time to build the full-text index $t$ remains in $O((W + R) \cdot log(W + R))$.

### 5.3.2 Retrieval

Since the `wordsfile` normally contains all words from all documents excluding stop words, using the `docsfile` to build the full-text index can result in more words in the text vocabulary. This can lead to more blocks, potentially slightly worsening query times. At the same time, words are not duplicated anymore, which can lead to smaller blocks, potentially slightly improving query times.

# 6. Text Index Block Changes

In chapter 3 we saw how the half-inverted full-text index is built. To recall, each word is assigned a `BlockIndex`. Each word having the same four-letter prefix gets the same `BlockIndex` assigned. For all shorter words, an own block is created. The `BlockIndexs` are assigned in lexicographical order. This ordering is necessary to implement the half-inverted index. Since naturally some prefixes appear more often then others, this leads to vastly different block sizes. For the scientists dataset, this can be seen in Figure 6.1.

The 10 most common prefixes in the scientists dataset

| Number of Results | Prefix |
|---|---|
| 91,824 | work |
| 80,856 | scie |
| 79,722 | publ |
| 78,069 | univ |
| 54,155 | comp |
| 51,967 | awar |
| 51,965 | inte |
| 48,689 | rese |
| 47,413 | book |
| 44,533 | phys |

Figure 6.1: The 10 most common prefixes in the scientists dataset. This information was calculated by executing a *WordScan* for each prefix 'aaaa' to 'zzzz' on the scientists text index.

Comparing the first and the tenth most common prefixes in Figure 6.1, we can already see a factor of two difference. This difference only gets larger when looking at even less used prefixes. Out of a possible $26^4$ = 456,976 prefixes, only 36,235 appear in the scientists dataset.

In the past, to efficiently use this block structure, the retrieval was limited to *WordScan*s with prefixes of a minimum size of four. This leads to fast queries since only one block has to be read, but it limits the user.

## 6.1 Feature changes

To solve the problems mentioned above, a configurable block size was implemented together with the possibility to execute *WordScan*s with arbitrary prefix lengths. To enable *WordScan*s with arbitrary prefix lengths, multiple blocks have to be read and merged. Since the blocks are already sorted, a sorted table merger was implemented to improve the speeds of this compared to appending and sorting the blocks. We will look at the changes made in two parts. The full-text index building part and the retrieval part.

### 6.1.1 Full-text Index Building

The now set block size makes calculating the block boundaries obsolete. Furthermore, the half-inverted full-text index table is now split into two tables. One for words and another for entities. An example for the words table can be seen in Figure 6.2, and an example for the entity table can be seen in Figure 6.3. It can be seen that the column for the `BlockIndex` was removed from both tables. Moreover, the column indicating whether the row belongs to a word or entity was removed since it is not needed anymore. The entity table contains a column for `WordVocabIndex`es that is used to track the co-occurrences of words with entities.

To build these two tables, the `wordsfile` and, optionally, the RDF vocabulary are parsed, and each text record is collected like before. All word postings of the text record are written to the word table. In the old version, the entity postings were written to the half-inverted full-text index table for each block touched by the words of the text record. Since we do not know the blocks the words will end up in now, we cannot deduplicate the postings at this point. Therefore, we have to add each entity posting to the entity table once for each word of the text record.

After collecting all text records, both tables are sorted by `WordVocabIndex`. Then the word table is parsed. The user-specified number of word postings are collected. Once the specified block size is reached, all entity postings up to the currently largest `WordVocabIndex` are collected. The collected posting lists are then sorted by `Text-RecordIndex`, `WordVocabIndex` or `VocabIndex`, and `Score`. Now the entity list is deduplicated. Afterward, both lists are written to the block in the same manner the old version did.

The set number of word postings per block introduces a new caveat. Word postings with the same `WordVocabIndex` now can appear in multiple blocks. Therefore, it is not enough to track the block boundaries with only the largest `WordVocabIndex` of each block. Instead, the smallest and largest `WordVocabIndex`es for each block are now saved. Note that it suffices to write the entities co-occurring with a certain `WordVocab-Index` in the first block the `WordVocabIndex` appears in. This is because during retrieval all entity lists of all blocks containing the `WordVocabIndex` are retrieved, which includes the first block; therefore, no information is lost.

Word table example:

| `WordVocab-Index` | `Text-Record-Index` | `Score` | Word |
|---|---|---|---|
| 0 | 2 | 0 | `:s:firstsentence` |
| 1 | 1 | 1 | `astronomer` |
| 1 | 2 | 0 | `astronomer` |
| 2 | 1 | 1 | `astronomy` |
| 2 | 2 | 0 | `astronomy` |
| 2 | 3 | 1 | `astronomy` |
| 2 | 4 | 1 | `astronomy` |
| 3 | 3 | 1 | `concentrates` |
| 3 | 4 | 1 | `concentrates` |
| 4 | 3 | 1 | `earth` |
| 4 | 4 | 1 | `earth` |
| 5 | 1 | 1 | `field` |
| 5 | 2 | 0 | `field` |
| 5 | 4 | 1 | `field` |
| 6 | 3 | 1 | `outside` |
| 6 | 4 | 1 | `outside` |
| 7 | 3 | 1 | `question` |
| 8 | 1 | 1 | `scientist` |
| 8 | 2 | 0 | `scientist` |
| 9 | 3 | 1 | `scope` |
| 9 | 4 | 1 | `scope` |
| 10 | 3 | 1 | `specific` |
| 11 | 3 | 1 | `studies` |
| 11 | 4 | 1 | `studies` |

Figure 6.2: Example for the new word table used in the full-text index building of the *Text Index Block Changes* feature.

### 6.1.2 Retrieval

The new retrieval works similar to the old retrieval except that now not only one block is returned for a given `WordVocabIndex` range but multiple. For both *WordScan*s and *EntityScan*s, we now do two binary searches. One to find the first block containing the

Entity table example:

| Word | Word-Vocab-Index | Vocab-Index | Text-Record-Index | Score | Entity |
|---|---|---|---|---|---|
| astronomer | 1 | 0 | 1 | 0 | <Astronomer> |
| astronomer | 1 | 0 | 2 | 0 | <Astronomer> |
| astronomy | 2 | 0 | 1 | 0 | <Astronomer> |
| astronomy | 2 | 0 | 2 | 0 | <Astronomer> |
| field | 5 | 0 | 1 | 0 | <Astronomer> |
| field | 5 | 0 | 2 | 0 | <Astronomer> |
| scientist | 8 | 0 | 1 | 0 | <Astronomer> |
| scientist | 8 | 0 | 2 | 0 | <Astronomer> |

Figure 6.3: Example for the new entity table used in the full-text index building of the *Text Index Block Changes* feature.

smaller `WordVocabIndex` of the queried range, and one to find the last block containing the larger `WordVocabIndex` of the queried range. After both blocks are found, they are collected together with all blocks that are in between. For a *WordScan*, all blocks are filtered by the queried `WordVocabIndex` range. Note, only the first and last block can contain `WordVocabIndex`es outside the queried range. After all blocks are collected, a sorted merge is performed. For a *WordScan*, only the `TextRecordIndex` sorting is preserved; for the *EntityScan*, the `TextRecordIndex` and `VocabIndex` sorting is preserved. The reason for this stronger sorting during the *EntityScan* is the removal of duplicate entity and text record combinations in the result, since multiple blocks could contain the same entity and text record combinations. Afterward, the same format of result is returned as in the old retrieval.

## 6.2 Theoretical Analysis

We will again look at the influence of the changes made to the runtime in two parts. The full-text index building part and the retrieval part.

### 6.2.1 Full-text Index Building

#### Calculating the Block Boundaries

It should be clear that the time needed to calculate the block boundaries is removed from the full-text index building. This does not change the asymptotic runtime of the total full-text index building.

### Building the half-inverted Text Index Table

In 3.5.1 we saw that for each text record all entities of the text record were written to the blocks touched by the words of the text record. We made the assumption that the number of touched blocks is proportional to the number of words. Now there are no pre-calculated blocks which means each entity of a text record is written to the entity table once for each word in the text record. Therefore the worst case for the number of writing operations done still is $(T_{max})^2$. This leads to the asymptotic runtime remaining the same for this step and still being in $O(W + R)$.

### Sorting the half-inverted Index Tables

First we will look at the runtime of sorting two tables, one with size $A$ and one with size $B$, compared to the runtime of sorting a table of size $A + B$. If $A, B > 0$:

$$\begin{aligned}
&A \cdot log(A) + B \cdot log(B) \\
&\leq A \cdot log(A + B) + B \cdot log(A + B) \\
&= (A + B) \cdot log(A + B)
\end{aligned}$$

The difference is:

$$\begin{aligned}
&(A + B) \cdot log(A + B) - (A \cdot log(A) + B \cdot log(B)) \\
&= A \cdot (log(A + B) - log(A)) + B \cdot (log(A + B) - log(B)) \\
&= A \cdot log(1 + \frac{B}{A}) + B \cdot log(1 + \frac{A}{B})
\end{aligned}$$

If $A \propto B$ this leads to the difference being in:

$$O(A)$$

But since $(A + B) \cdot log(A + B)$ asymptotically outgrows $A$ this does not change the asymptotic runtime. Since in our case the size of the word table is proportional to the size of the entity table this holds and the asymptotic runtime stays in $O((W + R) \cdot log(W + R))$.

### Writing the half-inverted Index Tables

This operation is now extended by the sorting of blocks before they are written. The number of blocks is proportional to $(W + R)$. For each block the size of the word list and the entity list are proportional to each other. Let $B_s$ be the number of word postings per block the user specified. Then the time to write the half-inverted index tables $t_7$ is:

$$t_7 \in O((W + R) \cdot B_s \cdot log(B_s))$$

The user specified size of the blocks can be viewed as constant leading to:

$$t_7 \in O(W + R)$$

### Overall Runtime

With the block changes made the asymptotic runtime for full-text index building does not change since $(W + R) \cdot log(W + R)$ still is the strongest growing term.

## 6.2.2 *WordScan*

### Finding the correct blocks

With the *Text Index Block Changes* feature multiple blocks have to be retrieved. This means two binary searches have to be performed to find the first and last block. This does not change the asymptotic runtime which is in $O(log(B))$ for the number of blocks in the full-text index $B$. Since with this feature the number of blocks is proportional to $(R + W)$ this leads to a runtime $t_1$:

$$t_1 \in O(log(R + W))$$

### Read the word list of all blocks

Now we know each block has a word list of size $B_s$. But we do not now the number of blocks returned. Let $P$ be the number of word postings matching the search term of the query. Then we know that at maximum $\lceil P/B_s \rceil + 1$ blocks are returned. The number of blocks is proportional to $P$ which leads to a runtime $t_2$:

$$t_2 \in O(B_s \cdot P)$$

If we view $B_s$ as constant this leads to:

$$t_2 \in O(P)$$

**Filter the intermediate result**

From all blocks returned only the first and the last can contain `WordVocabIndex`es out of the queried range. Since both block word lists are of size $B_s$, the runtime of this step $t_3$ is:

$$t_3 \in O(B_s)$$

If we view $B_s$ as constant this leads to:

$$t_3 \in O(1)$$

**Replcaing IDs by readable values**

The result size is proportional to $P$ which leads to a runtime $t_4$ of:

$$t_4 \in O(P)$$

**Over all time for a *WordScan***

This leads to a total runtime $t$ with:

$$t \in O(log(W + R) \cdot P)$$

### 6.2.3 *EntityScan*

The size of the entity list is proportional to $P$ as well as the size of the word list the entity list is joined with. This leads to the biggest asymptotic term being $P^2$ during the join of both lists. The details why this is where explained in 3.5.3. This leads to the runtime of an *EntityScan* $t \in O(log(W + R) \cdot P^2)$.

# 7. Empirical analysis

## 7.1 Setup

To perform empirical tests, a Hyper-V virtual machine with Ubuntu Server 24.04.3 LTS was used. The virtual machine runs on a Windows 11 PC with an Intel Core i5-11600K 3.90GHz CPU having 6 cores and 12 logical processors, an NVIDIA GeForce RTX 2070 Super GPU, 32 GB 3200 MT/s DDR4 RAM, and a Samsung SSD 980 PRO 1 TB SSD. The virtual machine has a set amount of 8 GB of RAM and 6 virtual cores.

## 7.2 Datasets

Two datasets were used run empirical tests. The scientists dataset and the yago-3 dataset. Since the yago-3 dataset comes in multiple turtle files, they were merged using the following command:

```
cat *.ttl > yagoComplete.ttl
```

For the scientists dataset, a `wordsfile` was created that only contains the entity mentions from the original `wordsfile` using the following command:

```
awk -F'\textbackslash t' '\$2 == 1' \
    scientists.wordsfile.tsv \
    > scientists.entityfile.tsv
```

## 7.3 Program Builds

For each of the three features, the IndexBuilderMain and the ServerMain with a baseline version and a feature version have been built. The baseline version is the commit the feature is based upon. This means the difference between the baseline version and the feature version is only the difference of the feature branch. One exception is the *Text Index Block Changes* feature. The difference between the baseline version and the feature version of the *Text Index Block Changes* feature is two branches. One branch implements the *Text Index Block Changes*, and the other implements the sorted table merge during retrieval. Also for *Text Index Block Changes*, there exists a baseline old that refers to QLever before the implementation of arbitrary prefix search.

The cmake configure command used:

```
cmake -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_MAKE_PROGRAM=ninja \
    -G Ninja -S . -B ./build \
    -DJEMALLOC_MANUALLY_INSTALLED=True \
    -DUSE_PARALLEL=true \
    -DSINGLE_TEST_BINARY=ON
```

The cmake build command used:

```
cmake --build ./build --target all -j 4
```

For the *Text Index Literal Filtering* baseline the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Thu Aug 28 04:44:55 PM UTC 2025"
-- GIT_HASH is "b46dbfd1e0f2470a31c0e79e6f6cee6c75d6d288"
```

For the *Text Index Literal Filtering* feature the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Wed Sep  3 06:00:17 PM UTC 2025"
-- GIT_HASH is "0699d9bf0463a214af8e14c0523af7b2803a2e43"
```

For the *Document only Text Index Building* baseline the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Thu Aug 28 04:44:55 PM UTC 2025"
-- GIT_HASH is "b46dbfd1e0f2470a31c0e79e6f6cee6c75d6d288"
```

For the *Document only Text Index Building* feature the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Thu Sep  4 05:38:40 PM UTC 2025"
-- GIT_HASH is "7355610780daf0187c8c54a84ed059830da20f1b"
```

For the *Text Index Block Changes* baseline old the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Fri Sep  5 09:40:12 AM UTC 2025"
-- GIT_HASH is "d85d630cb6e0b59396737771c31d73bc4cdcf359"
```

For the *Text Index Block Changes* baseline the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Tue Sep  2 10:29:56 PM UTC 2025"
-- GIT_HASH is "299804ed57c615c9f4e87c4c953788f7acd6b45b"
```

For the *Text Index Block Changes* feature the compilation info is:

```
-- DATETIME_OF_COMPILATION is "Tue Sep  2 10:08:44 PM UTC 2025"
-- GIT_HASH is "7f6395b1cb3e43bae46b3696e3b744a6fd149647"
```

## 7.4 Index Builds

The runtime of the index builds was measured using the 'time' command from GNU Time [11]. For each feature, there are commands with different index building options to compare the old QLever version with the new features. We will refer to a combination of options as configuration. Some options are the same across all configurations. The options '-i', '-f', and '-s' are dataset specific options. Since the 'yago-3' dataset was used to test the *Text Index Literal Filtering* feature, the option '-i' specifying the index name is set to 'yago-3'. Analogously, the 'scientists' dataset was used for the *Document only Text Index Building* and *Text Index Block Changes* features, leading to the index name 'scientists'. The same logic applies to the '-f' option for setting the RDF dataset file and the '-s' option for setting the settings file. There are two options that are the same across all configurations. The '-m' option for setting the amount of memory available for index building. This was set to 8 GB for all builds. The option '-p' specifying whether the index building should be parallelized was set to true for all configurations. Note that currently the full-text index building does not offer parallelization; therefore, this option only plays a crucial role in RDF index building.

As explained in 7.3, the baseline only differs from the feature in the changes made in this work. Since QLever's index builder provides the option to 'only add the full-text index' on top of an existing RDF index, this was tested as well. This leads to each configuration being tested by building both indexes and by only adding the full-text index. Each measurement of the 'only add full-text index' was executed on the RDF index previously built using the same configuration for building both indexes. The only exception to this is the *Text Index Literal Filtering* feature. Since the regex filter cannot be executed together with the option to 'only add a full-text index' it was replaced by the option to add all literals. Note this still leads to a filtered full-text index since the filtering happened during the complete index building before.

### 7.4.1 Text Index Literal Filtering

There are three different option configurations that were tested on the *Text Index Literal Filtering* feature.

1. Baseline build with add all literals

2. Feature build with add all literals

3. Feature build with filter 'hasGloss'

Configuration 1 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Baseline/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true -W \
-s $YAGO3/yago-3.settings.json -m 8GB
```

Configuration 1 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Baseline/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true -W \
-s $YAGO3/yago-3.settings.json -m 8GB -A
```

Configuration 2 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Feature/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true -W \
-s $YAGO3/yago-3.settings.json -m 8GB
```

Configuration 2 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Feature/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true -W \
-s $YAGO3/yago-3.settings.json -m 8GB -A
```

Configuration 3 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Feature/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true \
-r 'hasGloss' -s $YAGO3/yago-3.settings.json -m 8GB
```

Configuration 3 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/TextIndexLiteralFiltering/\
Feature/IndexBuilderMain \
-i yago-3 -f $YAGO3/yagoComplete.ttl -p true \
-W -s $YAGO3/yago-3.settings.json -m 8GB -A
```

### 7.4.2 Document only Text Index Building

There are four different configurations that were tested on the *Document only Text Index Building* feature:

1. Baseline build with options to use `wordsfile` and `docsfile` and add all literals
2. Feature build with options to use `wordsfile` and `docsfile` and add all literals
3. Feature build with options to only use `docsfile` and add all literals
4. Feature build with options to only add entities from `wordsfile`, using the pre built file explained in 7.2, together with words from the `docsfile` and add all literals

Configuration 1 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Baseline/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

Configuration 1 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Baseline/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -A
```

Configuration 2 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

Configuration 2 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -A
```

Configuration 3 complete index build::

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv -D -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

Configuration 3 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv -D -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -A
```

Configuration 4 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv -D -E \
-w $SCIENTIST/scientists.entityfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

Configuration 4 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/DocsFileOnlyTextIndex/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv -D -E \
-w $SCIENTIST/scientists.entityfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -A
```

### 7.4.3 Text Index Block Changes

There are three different configurations that were tested on the *Text Index Block Changes* feature. For completeness the configuration for the old baseline complete index build is also shown here but this was not timed.

1. Baseline build with options to use `wordsfile` and `docsfile` and add all literals
2. Feature build with options to use `wordsfile` and `docsfile` and add all literals with a set block size of 5,000
3. Feature build with options to use `wordsfile` and `docsfile` and add all literals with a set block size of 100,000

Configuration 1 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Baseline/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

Configuration 1 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Baseline/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -A
```

Configuration 1 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -P 5000
```

Configuration 2 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -P 5000 -A
```

Configuration 1 complete index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -P 100000
```

Configuration 3 only add text index build:

```
/usr/bin/time -v \
$QLEVER/qlever-builds/BlockChanges/\
Feature/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB -P 100000 -A
```

Complete index building with baseline old (untimed):

```
$QLEVER/qlever-builds/BlockChanges/\
BaselineOld/IndexBuilderMain \
-i scientists -f $SCIENTIST/scientists.nt -p true \
-d $SCIENTIST/scientists.docsfile.tsv \
-w $SCIENTIST/scientists.wordsfile.tsv -W \
-s $SCIENTIST/scientists.settings.json -m 8GB
```

## 7.5 Results

For each feature in this work, measurements were made for the index building as described in section 7.4. For each feature, this results in two tables containing measurements for index building. One table for the complete index builds. And one table for the 'only add full-text index' builds. The first column of each table corresponds to the configurations explained in section 7.4. Each table also has columns for the three measurements made, as well as for the mean, the variance, and the standard deviation.

Together with the measurement tables, there are t-test tables. The t-test tables are used to show whether the timing difference between two configurations is statistically relevant. The first column of the t-test tables shows which two configurations are compared. The second column shows the resulting p-value of the t-test, and the last column shows whether this p-value is smaller than 0.05. If the p-value is smaller than 0.05, this means the difference in times between the two configurations is statistically relevant.

Lastly, for each feature, there exists at least one example query executed on each index to show what difference the feature provides compared to before. The times for the query runtime have been measured with the QLever UI [1]. The QLever UI shows the time used to compute a given query. The cache has been cleared after each measurement.

### 7.5.1 Text Index Literal Filtering

The configurations were:

1. Baseline build with add all literals
2. Feature build with add all literals
3. Feature build with filter 'hasGloss'

#### Index Building Results

The timings for the complete index building can be seen in Figure 7.1, and the corresponding t-tests in Figure 7.2. The timings for the 'only add full-text index' builds can be seen in Figure 7.3, and the corresponding t-tests in Figure 7.4.

Text Index Literal Filtering Timing Results for complete index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 984 | 990 | 976 | 983.33 | 49.33 | 7.02 |
| 2 | 978 | 1004 | 979 | 987 | 217 | 14.73 |
| 3 | 378 | 393 | 394 | 388.33 | 80.33 | 8.96 |

Figure 7.1: *Text Index Literal Filtering* Timing Results for complete index builds. The timings are in seconds. The timings are rounded to the nearest second. The mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using all literals. Configuration 2 is the feature and builds using all literals. Configuration 3 is the feature and builds with the literal filter 'hasGloss'.

Text Index Literal Filtering Timing Results t-tests for complete index building

| Configuration Combination | p-value | (p<0.05) |
|---|---|---|
| 1 and 2 | 0.7242760 | false |
| 1 and 3 | 0.0000002 | true |
| 2 and 3 | 0.0000039 | true |

Figure 7.2: *Text Index Literal Filtering* Timing Results t-tests for complete index building. The p-values are rounded to seven decimal places. Configuration 1 is the baseline and builds using all literals. Configuration 2 is the feature and builds using all literals. Configuration 3 is the feature and builds with the literal filter 'hasGloss'.

Text Index Literal Filtering Timing Results for only add text index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 587.99 | 586.11 | 576.42 | 583.51 | 38.55 | 6.21 |
| 2 | 587.89 | 565.76 | 579.51 | 577.72 | 124.84 | 11.17 |
| 3 | 1.43 | 1.41 | 1.42 | 1.42 | 0.00 | 0.01 |

Figure 7.3: *Text Index Literal Filtering* Timing Results for only add text index builds. The timings are in seconds. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using all literals. Configuration 2 is the feature and builds using all literals. Configuration 3 is the feature and builds upon an index which was built with the literal filter 'hasGloss'.

| Text Index Literal Filtering Timing Results t-tests for only add text index building | | |
|---|---|---|
| Configuration Combination | p | (p<0.05) |
| 1 and 2 | 0.4880432 | false |
| 1 and 3 | 0.0000379 | true |
| 2 and 3 | 0.0001253 | true |

Figure 7.4: *Text Index Literal Filtering* Timing Results t-tests for only add text index building. The p-values are rounded to seven decimal places. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using all literals. Configuration 2 is the feature and builds using all literals. Configuration 3 is the feature and builds upon an index which was built with the literal filter 'hasGloss'.

### Test Queries

For the test queries, a *WordScan* query is chosen that scans for a number. Since numbers are part of many literals that hold no information for the full-text index, this shows the difference between the filtered and non-filtered version. An example of a common IRI used together with strings to form literals is the `http://yago-knowledge.org/resource/degrees` IRI. This is a 'yago-3' specific datatype which, leads to QLever categorizing values of it as literal. The query and the results on all three configurations can be seen in Figure 7.5 and Figure 7.6. The second test query can be seen in Figure 7.7. The query can be used to find predicates occurring together with literal objects containing the queried word. The results of this query when executed on an index built with configuration 2 can be seen in Figure 7.8.

*WordScan* query for a number:

```
SELECT * WHERE {
    ?t ql:contains-word "1986" .
}
LIMIT 5
```

Figure 7.5: A *WordScan* query scanning for a number.

Results of executing query seen in Figure 7.5 on all *Text Index Literal Filtering* configurations:

**Result for *Text Index Literal Filtering* configuration 1:**

| ?ql_score_word_t_1986 | ?t |
|---|---|
| 1 | - |
| 1 | - |
| 1 | - |
| 1 | - |
| 1 | - |

**Result for *Text Index Literal Filtering* configuration 2:**

| ?ql_score_word_t_1986 | ?t |
|---|---|
| 1 | -1.1986 |
| 1 | -105.1986 |
| 1 | -106.1986 |
| 1 | -11.1986 |
| 1 | -112.1986 |

**Result for *Text Index Literal Filtering* configuration 3:**

| ?ql_score_word_t_1986 | ?t |
|---|---|
| 1 | "a psychoactive drug . . . in 1986 . . . " |

Figure 7.6: Results of executing query seen in Figure 7.5 on all *Text Index Literal Filtering* configurations. The values in the '?t' column of the second configuration are of type `http://yago-knowledge.org/resource/degrees`. Configuration 1 is the baseline and builds using all literals. Configuration 2 is the feature and builds using all literals. Configuration 3 is the feature and builds with the literal filter 'hasGloss'.

Query to find what predicates occur together with literals containing the word "1986":

```
SELECT ?p (COUNT(?p) AS ?count) WHERE {
    ?t ql:contains-word "1986" .
    ?t ql:contains-entity ?e .
    ?s ?p ?e
}
GROUP BY ?p
ORDER BY DESC(?count)
```

Figure 7.7: Query to find what predicates occur together with literals containing the word "1986".

Results for the query shown in Figure 7.7 executed on the *Text Index Literal Filtering* configuration 2

| ?p | ?count |
|---|---|
| occursSince | 9918 |
| wasCreatedOnDate | 8355 |
| label | 6725 |
| wasBornOnDate | 5220 |
| diedOnDate | 3401 |
| occursUntil | 3329 |
| redirectedFrom | 2507 |
| prefLabel | 2014 |
| happenedOnDate | 776 |
| wasDestroyedOnDate | 407 |
| hasLongitude | 96 |
| hasLatitude | 64 |
| hasArea | 3 |
| startedOnDate | 2 |
| hasPopulationDensity | 1 |
| hasMotto | 1 |
| hasGloss | 1 |
| hasGeonamesEntityId | 1 |

Figure 7.8: Results of the query shown in Figure 7.7 executed on the *Text Index Literal Filtering* configuration 2.

### 7.5.2 Document only Text Index Building

The configurations were:

1. Baseline build with options to use `wordsfile` and `docsfile` and add all literals
2. Feature build with options to use `wordsfile` and `docsfile` and add all literals
3. Feature build with options to use words from the `docsfile` and add all literals
4. Feature build with options to only add entities from `wordsfile`, using the pre built file explained in 7.2, together with words from the `docsfile` and add all literals

#### Index Building Results

The timings for the complete index building can be seen in Figure 7.9, and the corresponding t-tests in Figure 7.10. The timings for the 'only add full-text index' builds can be seen in Figure 7.11, and the corresponding t-tests in Figure 7.12.

Document only Text Index Building Timing Results for complete index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 18.86 | 18.47 | 18.61 | 18.65 | 0.04 | 0.29 |
| 2 | 19.01 | 19.09 | 19.09 | 19.06 | 0.0 | 0.05 |
| 3 | 11.63 | 11.93 | 12.55 | 12.04 | 0.22 | 0.47 |
| 4 | 17.80 | 18.23 | 18.24 | 18.09 | 0.06 | 0.25 |

Figure 7.9: *Document only Text Index Building* Timing Results for complete index builds. The timings are in seconds. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

Document only Text Index Building Timing Results t-tests for complete index building

| Configuration Combination | p | (p<0.05) |
|---|---|---|
| 1 and 2 | 0.0606830 | false |
| 1 and 3 | 0.0003878 | true |
| 1 and 4 | 0.0421140 | true |
| 2 and 3 | 0.0013540 | true |
| 2 and 4 | 0.0187433 | true |
| 3 and 4 | 0.0002526 | true |

Figure 7.10: *Document only Text Index Building* Timing Results t-tests for complete index building. The p-values are rounded to seven decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

Document only Text Index Building Timing Results for only add text index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 16.50 | 17.62 | 17.07 | 17.06 | 0.31 | 0.56 |
| 2 | 18.14 | 18.04 | 18.24 | 18.14 | 0.01 | 0.10 |
| 3 | 11.11 | 11.40 | 11.19 | 11.23 | 0.02 | 0.15 |
| 4 | 16.06 | 16.03 | 15.99 | 16.03 | 0.00 | 0.04 |

Figure 7.11: *Document only Text Index Building* Timing Results for only add text index builds. The timings are in seconds. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

Document only Text Index Building Timing Results t-tests for only add text index building

| Configuration Combination | p | (p<0.05) |
|---|---|---|
| 1 and 2 | 0.0752769 | false |
| 1 and 3 | 0.0017702 | true |
| 1 and 4 | 0.0844799 | false |
| 2 and 3 | 0.0000015 | true |
| 2 and 4 | 0.0002129 | true |
| 3 and 4 | 0.0001664 | true |

Figure 7.12: *Document only Text Index Building* Timing Results t-tests for only add text index building. The p-values are rounded to seven decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

### Test Queries

For the test queries a *WordScan* on a stop word was chosen to highlight the words added by the `docsfile`. The second query chosen is an *EntityScan*. Query 1 and its results can be seen in Figure 7.13 and Figure 7.14. Query 2 and its results can be seen in Figure 7.15 and Figure 7.16.

*WordScan* for a stop word:

```
SELECT * WHERE {
    ?t ql:contains-word "is"
}
```

Figure 7.13: A *WordScan* query scanning for a stop word.

Results of executing query seen in Figure 7.13 on all *Document only Text Index Building* configurations:

| Configuration | Result Size and Time |
|---|---|
| 1 | Fails because no block containing "is" exists |
| 2 | Fails because no block containing "is" exists |
| 3 | Returns 54,261 results in 1ms |
| 4 | Returns 54,261 results in 1ms |

Figure 7.14: Results of executing query seen in Figure 7.13 on all *Document only Text Index Building* configurations. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

*EntityScan* Query:

```
SELECT * WHERE {
    ?t ql:contains-word "space" .
    ?t ql:contains-entity <Astronomer> .
} ORDER BY ?t
```

Figure 7.15: A simple *EntityScan* query.

Results of executing query seen in Figure 7.15 on all *Document only Text Index Building* configurations:

**Result for *Document only Text Index Building* configuration 1:**

| ?ql_score _word_t _space | ?t | ?ql_score_t _fixedEntity__60 _Astronomer_62_ |
|---|---|---|
| 1 | Karl Gordon Henize, Ph. D. . . . | 0 |
| 1 | Astronomer Michelle Thaller . . . | 0 |
| 1 | Astronomer Michelle Thaller . . . | 0 |
| 1 | In Russia, Gurshtein was active . . . | 0 |
| 1 | Michael C. Malin (born 1950) . . . | 0 |
| 1 | Michael C. Malin (born 1950) . . . | 0 |
| 2 | Michael C. Malin (born 1950) . . . | 0 |
| 1 | Rodger Evans Doxsey . . . | 0 |
| 1 | Rodger Evans Doxsey . . . | 0 |

**Result for *Document only Text Index Building* configuration 3:**

| ?ql_score _word_t _space | ?t | ?ql_score_t _fixedEntity__60 _Astronomer_62_ |
|---|---|---|
| | | |

**Result for *Document only Text Index Building* configuration 4:**

| ?ql_score _word_t _space | ?t | ?ql_score_t _fixedEntity__60 _Astronomer_62_ |
|---|---|---|
| 0 | Karl Gordon Henize, Ph. D. . . . | 0 |
| 0 | David C. Jewitt (born 1958) . . . | 0 |
| 0 | Spencer Jones's successor . . . | 0 |
| 0 | Woolley is known for his . . . | 0 |
| 0 | On appointment as Astronomer . . . | 0 |
| 0 | " Anyone ", said Terry and . . . | 0 |
| 0 | Astronomer Michelle Thaller . . . | 0 |
| 0 | In Russia, Gurshtein was active . . . | 0 |
| 0 | Michael C. Malin (born 1950) . . . | 0 |
| 0 | Rodger Evans Doxsey . . . | 0 |
| 0 | Alexander (Sasha) Kashlinsky . . . | 0 |

Figure 7.16: Results of executing query seen in Figure 7.15 on all *Document only Text Index Building* configurations. Configuration 2 is not shown since it yields the same result as configuration 1 and does not change anything. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 3 is the feature and builds using only the `docsfile`. Configuration 4 is the feature and builds using words from the `docsfile` and entities from the `wordsfile`.

### 7.5.3 Text Index Block Changes

The configurations were:

1. Baseline build with options to use `wordsfile` and `docsfile` and add all literals
2. Feature build with options to use `wordsfile` and `docsfile` and add all literals with a set block size of 5,000
3. Feature build with options to use `wordsfile` and `docsfile` and add all literals with a set block size of 100,000
4. Baseline build on the old version of QLever where short prefix scans were not yet available. Build with options to use `wordsfile` and `docsfile` and add all literals. This configuration was not timed but is tested in the test queries.

#### Index Building Results

The timings for the complete index building can be seen in Figure 7.17, and the corresponding t-tests in Figure 7.18. The timings for the 'only add full-text index' builds can be seen in Figure 7.19, and the corresponding t-tests in Figure 7.20.

Text Index Block Changes Timing Results for complete index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 18.93 | 18.96 | 18.09 | 18.66 | 0.24 | 0.49 |
| 2 | 17.15 | 17.27 | 17.88 | 17.43 | 0.15 | 0.39 |
| 3 | 16.84 | 17.39 | 17.12 | 17.12 | 0.08 | 0.28 |

Figure 7.17: *Text Index Block Changes* Timing Results for complete index builds. The timings are in second. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000.

Text Index Block Changes Timing Results t-tests for complete index building

| Configuration Combination | p | (p<0.05) |
|---|---|---|
| 1 and 2 | 0.0302771 | true |
| 1 and 3 | 0.0162591 | true |
| 2 and 3 | 0.3222605 | false |

Figure 7.18: *Text Index Block Changes* Timing Results t-tests for complete index building. The p-values are rounded to seven decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000.

Text Index Block Changes Timing Results for only add text index builds

| Con | T1 | T2 | T3 | Mean | Var | Dev |
|---|---|---|---|---|---|---|
| 1 | 17.01 | 17.35 | 17.24 | 17.20 | 0.03 | 0.17 |
| 2 | 15.48 | 16.09 | 15.52 | 15.70 | 0.12 | 0.34 |
| 3 | 14.85 | 14.32 | 14.12 | 14.43 | 0.14 | 0.38 |

Figure 7.19: *Text Index Block Changes* Timing Results for only add text index builds. The timings are in second. The timings as well as the mean, variance and standard deviation are rounded to two decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000.

| Text Index Block Changes Timing Results t-tests for only add text index building | | |
|---|---|---|
| Configuration Combination | p | (p<0.05) |
| 1 and 2 | 0.0067131 | true |
| 1 and 3 | 0.0018782 | true |
| 2 and 3 | 0.0127885 | true |

Figure 7.20: *Text Index Block Changes* Timing Results t-tests for only add text index building. The p-values are rounded to seven decimal places. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000.

### Test Queries

For the test queries a *WordScan* on the short prefix "a*" was chosen. The second query chosen is a *WordScan* on the word "workaholic" which is a rare word in the largest prefix block of the old full-text index. Query 1 and its results can be seen in Figure 7.21 and Figure 7.22. Query 2 and its results can be seen in Figure 7.23 and Figure 7.23.

*WordScan* for a short prefix:

```
SELECT * WHERE {
    ?t ql:contains-word "a*" .
}
```

Figure 7.21: A *WordScan* query scanning for a short prefix.

Results of executing query seen in Figure 7.21 on all *Text Index Block Changes* configurations:

**Timing results for query:**

| Con | T1 | T2 | T3 | T4 | T5 | Mean |
|---|---|---|---|---|---|---|
| 1 | 51 | 61 | 61 | 55 | 57 | 57 |
| 2 | 80 | 80 | 83 | 80 | 73 | 79.2 |
| 3 | 61 | 53 | 51 | 53 | 53 | 54.2 |
| 4 | Fails since prefix is to short | | | | | |

**T-test for timings:**

| Configuration Combination | p-value | (p<0.05) |
|---|---|---|
| 1 and 2 | 0.0000242 | true |
| 1 and 3 | 0.3090787 | false |
| 2 and 3 | 0.0000065 | true |

Figure 7.22: Results of executing query seen in Figure 7.21 on all *Text Index Block Changes* configurations. The times are measured in milliseconds. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000. Configuration 4 is the old baseline that did not have arbitrary prefix length for *WordScan*s.

*WordScan* for a rare word beginning with a common four-letter prefix:

```
SELECT * WHERE {
    ?t ql:contains-word "workaholic" .
}
```

Figure 7.23: A *WordScan* query for a rare word beginning with a common four-letter prefix.

Results of executing query seen in Figure 7.23 on all *Text Index Block Changes* configurations:

| Configuration | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 |

Figure 7.24: Results of executing query seen in Figure 7.23 on all *Text Index Block Changes* configurations. The times are measured in milliseconds. Configuration 1 is the baseline and builds using the `wordsfile`, the `docsfile` and all literals. Configuration 2 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 5,000. Configuration 3 is the feature and builds using the `wordsfile`, the `docsfile` and all literals together with a full-text index block size of 100,000. Configuration 4 is the old baseline that did not have arbitrary prefix length for *WordScan*s.

# 8. Discussion

The discussion is split up into three parts. One part for each feature. In these parts the results seen in section 7.5 are evaluated.

## 8.1 Text Index Literal Filtering

The *Text Index Literal Filtering* feature had three goals. The first was to remove the iteration over the whole RDF vocabulary during full-text index building. The second was the possibility to show literals if they are returned as results of a full-text index scan. The third goal was the improvement of result quality due to filtering literals that provide no information for the full-text index.

In Figure 7.2 and Figure 7.4 we see that with the same options, the feature build does not provide a runtime advantage to the baseline build when it comes to index building. This changes drastically if we use a regex filter to reduce the number of literals in the full-text index. In Figure 7.1, we can see the time the complete index building takes with filtered literals is less than half compared to the time without filtering. The improvement is even more extreme if we only look at the 'only add text index' builds seen in Figure 7.3. The 'only add text index' build with filtered literals only takes 1/400 compared to adding all literals.

The results of evaluating the query seen in Figure 7.5 consisting of a *WordScan* on "1986" on the old QLever index can be seen in Figure 7.6. The results showed that while the engine found text records containing the word "1986", it could not show the literals the word occurred in. Building the full-text index with all literals with the new feature leads to the literals containing "1986" being shown, but we see the top results are all uninformative. Executing the query without a limit leads to 8,918 results. Most of these results are uninformative, which can be seen with a query showing the predicates appearing with literal objects that contain "1986" shown in Figure 7.7. The result of this query can be seen in Figure 7.8 and shows that "1986" appears in many literals following predicates that hint at uninformative literals. Especially the top 5 predicates hint at literals that are all only one sentence long. It can also be seen that the predicate 'hasGloss' occurs exactly once with a literal. Therefore, executing the query for "1986" on the filtered full-text index returns exactly on result which can be seen in Figure 7.6. While this is the only result, the text record consists of multiple sentences.

To conclude, the *Text Index Literal Filtering* feature without filtering already provides a benefit for QLever since it can retrieve the literals of full-text index scan queries. While the feature did not provide the expected runtime improvements when adding all literals, it certainly did when filtering for specific literals. Together with this, it could be seen that choosing the right filter together with a suitable dataset improves result quality.

## 8.2 Document only Text Index Building

The *Document only Text Index Building* feature had the goal of simplifying the full-text index building while keeping the functionality of a linked RDF and full-text index.

Comparing the runtime of the different index builds shows interesting results. In Figure 7.9 and Figure 7.11, it can be seen that, using the same options, the old QLever version seems to be a bit faster, even though this is not statistically backed up by the t-tests seen in Figure 7.10 and Figure 7.12. The reasons for this are unclear. One theory is a potential increase in one of the constant runtime costs. It can also be seen that building using only the `docsfile` is much faster than the other configurations. This is expected since not iterating the large `wordsfile` and not adding co-occurring entities to the full-text index saves time. At the same time, this leads to *EntityScan*s not functioning anymore, as seen in Figure 7.16. When comparing configuration 1 with configuration 4, it can be seen that configuration 4 seems slightly faster. This is statistically backed up for the complete index building as seen in Figure 7.10 but not for the 'only add full-text index' building as seen in Figure 7.12. A theory for this slight improvement in time could be the total number of words in the `docsfile` being less than the total number of words in the `wordsfile`. While the `docsfile` contains stop words, the `wordsfile` splits up documents into smaller, overlapping text records. The overlapping leads to the repeating of words, potentially increasing the number of words in the `wordsfile` beyond the total number of words in the `docsfile`.

When looking at the query results in Figure 7.14 for the query shown in Figure 7.13 consisting of a simple *WordScan* on the stop word "is", it can be seen that building the full-text index with the `docsfile` leads to stop words cluttering the full-text index. While this may be wanted for some in general, this increases the full-text index size without providing a benefit. When looking at the query results in Figure 7.16 for the query shown in Figure 7.15 consisting of a simple *EntityScan*, an interesting phenomenon can be seen that was explained earlier. Executed on configuration 1, the query returns duplicate results, and compared to configuration 4, there are some results missing. If we look at one text appearing for configuration 4 but not for configuration 1, we can see why this happens. For example, the text: “Spencer Jones’s successor as Astronomer Royal was Richard Woolley, who on taking up the position in 1956 responded to a question from the press and was misquoted as saying " Space travel is utter bilge "”. It can be seen that the word distance between "Astronomer" and "Space" is quite big. Since to build the `wordsfile`, each document was split up into smaller text records; the entity "<Astronomer>" does not appear in the same text record as the word "Space". The same pattern holds for all text records that appear in the result for configuration 4 but not in the result for configuration 1. The reason they appear for configuration 4 is that using the `wordsfile` to only add entities, each entity belonging to a text record is assigned to the document the text record originated from. For example, if text records 1 to 4 all came from the document with `DocumentIndex` 4, all entities appearing in those text records will be saved as appearing in text record 4 when only adding entities from the `wordsfile`. This is the same reason why, for configuration 1, multiple texts repeat. If, for example, text records 1 and 2 are returned and they both originated from the same document, this document will be shown for both of them when looking up their `TextRecordIndex`es in the 'docsDB'.

Concluding, it can be seen that building the full-text index using only the `docsfile` rarely ever makes sense. But building the full-text index using only the `docsfile` together with only the entities from the `wordsfile` provides an easier input format for the text dataset. In this format, no overlapping text records for documents appear, and it is simpler for the user to create their own text dataset while keeping the connection between the RDF and full-text index. One disadvantage of this feature is the increased size of the full-text index due to adding stop words.

## 8.3 Text Index Block Changes

The *Text Index Block Changes* feature had the goal of enabling prefix search for prefixes shorter than four. It also had the goal of even the block sizes and therefore removing the calculation of block boundaries. This should theoretically improve full-text index building times as well as query times for specific words in previously large blocks.

The timing results for the index building seen in Figure 7.17 and in Figure 7.19 show that full-text index building with the feature is faster when choosing block sizes of 5,000 or 100,000 compared to the baseline. The difference is statistically backed up, as seen in Figure 7.18 and in Figure 7.20. Two potential reasons for this speedup are the time saved by not calculating the block boundaries. And the time saved by sorting the word table and entity table separately, since sorting two smaller tables is faster than sorting one large table, as seen in section 6.2. This is especially the case since both tables have fewer columns compared to the old half-inverted index table, which reduces the cost of comparing two rows. Comparing the mean time of the index building, it can also be seen that building with blocks of size 5,000 is slower than building with blocks of size 100,000.

For the complete index building, this is not statistically backed up as seen in Figure 7.18 but for the 'only add full-text index' build it is statistically backed up as seen in Figure 7.20. There are potential reasons for a big block size being faster than a small block size when building the full-text index. A larger batch size when writing the blocks to disk as well as more consecutive steps when iterating the word table and the entity table could lead to better cache locality.

The timing results seen in Figure 7.22 for the query seen in Figure 7.21 consisting of a *WordScan* on a short prefix executed on all configurations show two things. First, it can be seen that the old QLever index could not return a result for a prefix shorter than four. Second, merging multiple sorted small blocks is slower than filtering one large block, even though both of them are in $O(n)$. Note that configuration 1 does not do a sorted merge when returning multiple sorted blocks but instead appends them into a large table and sorts this table. It can be seen that this is not statistically slower than configuration 3, which uses a sorted merge to combine the sorted blocks. Executing a query consisting of a simple *WordScan* seen in Figure 7.23 on every configuration leads to the results seen in Figure 7.23. Since the simple *WordScan* can be executed in one millisecond or less, the measured values are inaccurate. But one thing can still be seen. Since the time to compute is rounded to the nearest millisecond, the configuration with the block size of 5,000 seems to be a bit faster than the others. This would only be reasonable since filtering a small block for a single word is faster than filtering a big block. In reality, this improvement is not relevant.

In conclusion, the *Text Index Block Changes* feature simplified the full-text index building internally. It does provide a slight improvement when it comes to the index building times, and for a suitable block size, the query times are similar to the baseline. Compared with the old QLever full-text index, this feature now allows for a prefix search of arbitrary lengths while giving the user the ability to choose a block size fitting their dataset.

# 9. Future work

While the three changes improved usability, they also provide a foundation for future improvements. We will first look at possible improvements to the individual features before looking at a possible improvement for the full-text index building at the end.

## 9.1 Text Index Literal Filtering

There are two main things to consider for the future of this feature. Firstly, for now the filter only works on the predicates of triples. And secondly, the interleaving of the RDF index building with the full-text index building can be confusing and can lead to unexpected behavior for the user.

To combat the first problem, the filter is implemented in a way that is easily changeable to check any part of the triple. If implemented, the filter can be used on objects to filter certain languages of literals. It could then also filter certain literals containing specific IRIs.

One clever solution for the second problem would be running a classic SPARQL query during full-text index building that returns all `VocabIndex`es of certain filtered literals. This would add the time of this query to the time the full-text index building takes but would cleanly separate the RDF index building and full-text index building again.

## 9.2 Document only Text Index Building

A possible extension of the *Document only Text Index Building* feature would be a customizable tokenizer. Currently the documents and literals are split using a tokenizer that splits texts at non-alphanumeric characters. This is a rather basic approach and does not skip stop words. One idea would be a configurable minimal length tokens need. This would filter out some stop words from the full-text index, making it smaller potentially improving query times.

## 9.3 Text Index Text Index Block Changes

The *Text Index Block Changes* feature leads to an arbitrary length for prefixes in a *Word-Scan* and slightly improved index building times. But when it comes to retrieval times, it could be seen that the merging of multiple sorted blocks is not much faster than appending and sorting them. Furthermore, this merging happens in memory since blocks cannot yet be read lazily from disk. Approaching these two problems could further enhance this feature. Moreover, the now arbitrary number of retrieved blocks could be used to implement a new full-text index scan type similar to a *WordScan*. The scan could take a starting word or prefix and an ending word or prefix and return all text records that contain words in this range.

## 9.4 Text Index Building

The building of the RDF index in QLever is parallelized. This is not the case for the full-text index building. For building the text vocabulary, this parallelization could be done similar to the building of the RDF vocabulary. This means building partial vocabularies, which can later be merged into one text vocabulary. A similar thing could be done to build the half-inverted full-text index. For the half-inverted full-text index, it is important that each word and entity from a text record is collected in the same partial index. Since the `wordsfile` and `docsfile` are sorted by text record, a parallelized parser could be implemented. The sorted partial half-inverted indexes can then be merged into the complete half-inverted index and written to disk.

## 9.5 Outlook

While the RDF side of QLever was steadily improved, the full-text side largely remained untouched. This leads to a difference in quality, especially when it comes to the index building times since building the RDF index is much faster compared to building the full-text index. The changes made in this work extend the usability of QLever's full-text index and lay the groundwork for further improvements strengthening the position of QLever as SPARQL+Text engine.

# Bibliography

[1] *ad-freiburg/qlever-ui*. original-date: 2018-07-29T12:22:40Z. 2025-08-14. URL: https://github.com/ad-freiburg/qlever-ui (visited on 2025-09-07).

[2] *Apache Jena - Jena Full Text Search*. URL: https://jena.apache.org/documentation/query/text-query.html (visited on 2025-09-08).

[3] *apache/jena*. original-date: 2013-01-04T08:00:32Z. 2025-09-10. URL: https://github.com/apache/jena (visited on 2025-09-10).

[4] Hannah Bast and Björn Buchhold. "An index for efficient semantic full-text search". In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. CIKM'13: 22nd ACM International Conference on Information and Knowledge Management. San Francisco California USA: ACM, 2013-10-27, pp. 369–378. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505689. URL: https://dl.acm.org/doi/10.1145/2505515.2505689 (visited on 2025-09-10).

[5] 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 '17: ACM Conference on Information and Knowledge Management. Singapore Singapore: ACM, 2017-11-06, pp. 647–656. ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3132921. URL: https://dl.acm.org/doi/10.1145/3132847.3132921 (visited on 2025-07-30).

[6] Hannah Bast and Elmar Haussmann. "Open Information Extraction via Contextual Sentence Decomposition". In: *2013 IEEE Seventh International Conference on Semantic Computing*. 2013 IEEE Seventh International Conference on Semantic Computing (ICSC). Irvine, CA, USA: IEEE, 2013-09, pp. 154–159. ISBN: 978-0-7695-5119-7. DOI: 10.1109/ICSC.2013.36. URL: http://ieeexplore.ieee.org/document/6693511/ (visited on 2025-08-17).

[7] Holger Bast and Ingmar Weber. "Type less, find more: fast autocompletion search with a succinct index". In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. SIGIR06: The 29th Annual International SIGIR Conference. Seattle Washington USA: ACM, 2006-08-06, pp. 364–371. ISBN: 978-1-59593-369-0. DOI: 10.1145/1148170.

1148234. URL: https://dl.acm.org/doi/10.1145/1148170.1148234 (visited on 2025-09-10).

[8] Andrzej Białecki et al. “Apache lucene 4”. In: *SIGIR 2012 workshop on open source information retrieval*. sn. 2012, p. 17.

[9] Orri Erling and Ivan Mikhailov. “RDF Support in the Virtuoso DBMS”. In: *Networked Knowledge - Networked Media*. Ed. by Tassilo Pellegrini et al. Red. by Janusz Kacprzyk. Vol. 221. Series Title: Studies in Computational Intelligence. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 7–24. ISBN: 978-3-642-02183-1 978-3-642-02184-8. DOI: 10.1007/978-3-642-02184-8_2. URL: http://link.springer.com/10.1007/978-3-642-02184-8_2 (visited on 2025-09-10).

[10] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. USA: Academic Press, Inc., 1978. ISBN: 978-0-12-335750-2.

[11] David Keppel, David MacKenzie, and Assaf Gordon. *GNU Time*. Free Software Foundation, Inc. 2018.

[12] *RDF 1.1 Concepts and Abstract Syntax*. URL: https://www.w3.org/TR/rdf11-concepts/ (visited on 2025-08-29).

[13] *What is GraphDB? — GraphDB 11.1 documentation*. URL: https://graphdb.ontotext.com/documentation/11.1/ (visited on 2025-09-10).