QLever: A Query Engine for Efficient SPARQL+Text Search

Hannah Bast
University of Freiburg
79110 Freiburg, Germany
bast@cs.uni-freiburg.de

Björn Buchhold
University of Freiburg
79110 Freiburg, Germany
buchhold@cs.uni-freiburg.de

ABSTRACT

We present QLever, a query engine for efficient combined search on a knowledge base and a text corpus, in which named entities from the knowledge base have been identified (that is, recognized and disambiguated). The query language is SPARQL extended by two QLever-specific predicates *ql:contains-entity* and *ql:contains-word*, which can express the occurrence of an entity or word (the object of the predicate) in a text record (the subject of the predicate). We evaluate QLever on two large datasets, including FACC (the ClueWeb12 corpus linked to Freebase). We compare against three state-of-the-art query engines for knowledge bases with varying support for text search: RDF-3X, Virtuoso, Broccoli. Query times are competitive and often faster on the pure SPARQL queries, and several orders of magnitude faster on the SPARQL+Text queries. Index size is larger for pure SPARQL queries, but smaller for SPARQL+Text queries.

CCS CONCEPTS

•**Information systems** →**Database query processing; Query planning; Search engine indexing;** *Retrieval efficiency;*

KEYWORDS

SPARQL+Text; Efficiency; Indexing

1 INTRODUCTION

This paper is about efficient search in a knowledge base combined with text. For the purpose of this paper, a knowledge base is a collection of subject-predicate-object triples, where consistent identifiers are used for the same entities. For example, here are three triples from Freebase[1], the world's largest open general-purpose knowledge base, which we also use in our experiments:

<Neil Armstrong> <is-a> <Astronaut>
<Neil Armstrong> <nationality> <American>
<Neil Armstrong> <books-written> "First on the moon"

A knowledge base enables queries that express the search intent precisely. For example, using SPARQL (the de facto standard query language for knowledge bases), we can easily search for all astronauts and their nationalities as follows:

SELECT ?x ?y WHERE {
?x <is-a> <Astronaut> .
?x <nationality> ?y
} ORDER BY ASC(?x) LIMIT 100

The result is a flat list of tuples ?x ?y, where ?x is an astronaut and ?y their nationality. The *ORDER BY ASC(?x)* clause causes the results to be listed in ascending (lexicographic) order. The *LIMIT 100* clause limits the result to the first 100 tuples. Note that if an astronaut has k nationalities, they would contribute k tuples to the result. Also note that the triples in the body of the query can contain variables which are not specified as an argument of the *SELECT* operator and which are hence not shown in the result.

Keyword search in object strings. Knowledge bases can have arbitrary string literals as objects. See the third triple in the example above, where the object names the title of a book. SPARQL allows regular expression matches for such literals. Commercial SPARQL engines also offer keyword search in literals. For Virtuoso (described in Section 2), this is realized via a special predicate *bif:contains*, where the *bif* prefix stands for *built-in function*. For example, the following query searches for astronauts who have written a book with the words *first* and *moon* in the title:

SELECT ?x ?y WHERE {
?x <is-a> <Astronaut> .
?x <books-written> ?w .
?w bif:contains "first AND moon"
}

Fully combined SPARQL+Text search. For our query engine, we consider the following deeper integration of a knowledge base with text. We assume that the text is given as a separate corpus, and that named entity recognition and disambiguation (of the entities from the knowledge base in the text) has been performed. That is, each mention of an entity of the knowledge base in the text has been annotated with the unique ID of that entity in the knowledge base. For example, the well-known FACC [13] dataset (which we also use in our experiments in Section 5) provides such an annotation of the ClueWeb12 corpus with the entities from Freebase. Here is an example sentence from ClueWeb12 with one recognized entity from Freebase (note how the entity is not necessarily referred to with its full name in the text):

On July 20, 1969, Armstrong<Neil Armstrong> became the first human being to walk on the moon

With a knowledge base and a text corpus linked in this way, queries of the following kind are possible:

[1] In our examples, we actually use Freebase Easy [4], a sanitized version of Freebase with human-readable entity names. In the original Freebase, entity identifiers are alphanumeric, and human-readable names are available via an explicit *name* predicate.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'17, November 6–10, 2017, Singapore.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: https://doi.org/10.1145/3132847.3132921

```
SELECT ?x TEXT(?t) WHERE {
  ?x <is-a> <Astronaut> .
  ?t ql:contains-entity ?x .
  ?t ql:contains-word "walk moon"
} ORDER BY DESC(SCORE(?t)) TEXTLIMIT 1 LIMIT 16
```

This query finds astronauts that have a mention in a text record that also contains the words *walk* and *moon* (note that, for the sake of brevity, *ql:contains-word* does not require an AND like *bif:contains*). For our evaluation and all our examples in this paper, the text records are sentences.[2]

The *TEXT(?t)* adds the text record that mentions the entity and the two words as one component of each result tuple. The *ORDER BY DESC(SCORE(?t))* causes the entities to be ranked by the number of matching text records.[3] The *TEXTLIMIT* operator limits the number of distinct items per text record variable to be included in the result. For the query above, we thus get 16 result tuples consisting of an astronaut and one matching text record each, and we get those 16 astronauts with the largest number of matching text records. We call queries like the above SPARQL+Text queries in this paper.

SPARQL+Text queries allow for powerful search capabilities. For example, on Freebase+ClueWeb, the query above contains the 12 astronauts who actually walked on the moon (because many sentences in the ClueWeb12 corpus mention that they did). Such querying capabilities have been discussed and implemented in relatively few systems so far; a good overview is given in [6, Section 4.6], a recent survey on the broad field of semantic search. In Section 5, we compare ourselves against the fastest existing such system, as well as two other systems.

An important application of SPARQL+Text queries is *Question Answering*. One classical and much researched QA problem is to translate natural language queries to SPARQL queries; see [6, Section 4.8]. These systems work by generating candidate queries, and rank them by their degree of "semantic match" to the natural language query. In the process, often thousands of SPARQL queries have to be processed for each natural-language query. The faster these queries can be processed, the more candidates the system can consider. The most recent systems have moved to querying a knowledge base linked to text exactly in the way described above, for example [15], who also work on Freebase+ClueWeb. Such systems need to process large numbers of SPARQL+Text queries.

1.1 Contributions

This paper describes and evaluates a system that can efficiently process SPARQL+Text queries even on large datasets like Freebase+ClueWeb.[4] We consider the following as our main contributions:

- A query engine, called QLever, for a knowledge base linked to a text corpus as described above, which efficiently supports the predicates *ql:contains-word* and *ql:contains-entity* and which is also efficient (and competitive) for pure SPARQL queries.
- A new benchmark of 159 SPARQL+Text queries from 12 categories, including pure SPARQL queries and real-world queries adapted from the SemSearch Challenge [8].
- A performance evaluation on this benchmark, with a comparison against three state-of-the-art query engines. QLever's query times are fastest for the pure SPARQL queries *and* fastest by several orders of magnitude for the SPARQL+Text queries. On the large Freebase+ClueWeb dataset, QLever achieves subsecond query times even for complex SPARQL+Text queries; see Section 5.
- Efficient support for convenient text-search features, including: text snippets as part of the result (see the *TEXT(?t)* above), scores for ranking (see the *SCORE(?t)* above), and a TEXTLIMIT operator for limiting the number of text snippets.
- Technical contributions are: index layouts for efficient scan and text operations; query planning for SPARQL+Text queries based on dynamic programming; novel heuristics for result size estimation.
- All code, benchmark queries and our dataset are open source and available on GitHub (https://github.com/Buchhold/QLever). QLever can easily be set up for any knowledge base linked to a text corpus as described above.

[2] Text records could also be paragraphs, whole documents, or parts of sentences. Of course, like in text search, the search result depends on the unit of text chosen.

[3] Thus, *SCORE(?t)* is currently just a shorthand for *COUNT(*)* in combination with *GROUP BY ?t*. Customized ranking functions can be implemented, too.

[4] The important aspect of search quality is out of scope for this paper.

2 RELATED WORK

We distinguish three lines of related work: (1) other systems for search on a knowledge base (KB) linked with a text corpus; (2) SPARQL engines, with and without (limited) text-search capabilities; (3) systems for searching the Semantic Web, which solve a related, yet different problem. As mentioned above, this paper is concerned with efficient indexing and querying. Concerning the important aspect of the *quality* of these kinds of search, see the overview in [6, Section 4.6].

2.1 Systems for a KB linked to a text corpus

Three early systems for combined search on a knowledge base linked to a text corpus are KIM [18], Mímir [19], and ESTER [7]. They all yield document-centric instead of entity-centric results and are not suited for processing general SPARQL queries. Also, they are efficient only for very specific subclasses of queries.

Broccoli [5] is a follow-up work to ESTER which yields entity-centric results. It was designed for interactive, incremental query construction and supports a subset of SPARQL, namely tree-like queries with exactly one variable in the SELECT clause. Broccoli has no query planner and a simplistic KB index. We compare against Broccoli (for a subset of our benchmark) in Section 5.

2.2 SPARQL engines

SPARQL engines can be used for search on a KB combined with text in two ways: either by adding all co-occurrence information as triples to the KB or by adding triples with text literals as object and then using non-standard text-search features that are provided by some SPARQL engines. In Section 5, we compare against both variants, using RDF-3X for the first variant and Virtuoso for the second. Both systems are briefly described in the following, along with two systems for more specific use cases.

A fundamental idea for tailor-made indices for SPARQL engines is to store six copies of the data (triples), each sorted according to one of the six possible permutations of subject, predicate and object

(SPO, SOP, PSO, SOP, OSP, OPS). The first published use of this idea was for Hexastore [20] and RDF-3X [17]. Our engine, QLever, also makes use of this principle. In our evaluation, we use RDF-3X because its code is publicly available.

RDF-3X has an advanced query planner, which, in particular, finds the optimal join order for commonly used query patterns like star-shaped queries. Query execution of RDF-3X is pipelined, that is, joins can start before the full input is available. This is further accelerated by a runtime technique called sideways information passing (SIP): this allows multiple scans or joins with common columns in their input to exchange information about which segments in these columns can be skipped. QLever also has an advanced query planner but forgoes pipelining and SIP in favor of highly optimized basic operations and caching of sub-results.

SPARQL queries can be rewritten to SQL [12] and all the big RDBMSs now also provide support for SPARQL. A good representative of such a system is Virtuoso[5], because it is widely used in practice and in many SPARQL performance evaluations. It is built on top of its own full-featured relational database and provides both a SQL and a SPARQL front-end. Since Version 7, triples are stored column-wise and indexed in a way corresponding to the two permutations PSO and POS explained above. For queries involving predicate variables, there are additional indices and more permutations can be built on demand, thus boosting efficiency on such queries at the cost of increasing the index size. Virtuoso supports full-text search via its *bif:contains* predicate, explained in Section 1. This functionality is realized via a standard inverted index and allows keywords to match literals from the knowledge base. The same approach is used by other SPARQL engines with support for keyword search, for example, in Jena (see http://jena.apache.org/documentation/query/text-query.html). As explained in Section 1, this does not support entity occurrences anywhere in the text like QLever's *ql:contains-entity* predicate does.

There is also work on SPARQL engines with data layouts that are tuned towards specific query patterns. One basic idea in this context are property tables, where data that is often accessed together is put in the same table (for example, a table for books, with one row per book containing its ID, title, and year of publication). In [21], joins between such tables are expedited by precomputing for each URI its occurrences in all property tables. These approaches require prior knowledge of typical query patterns. Since we explore general-purpose SPARQL+Text search, we have not included these systems in our evaluation.

2.3 Semantic Web Search

The contents of the Semantic Web can be viewed as a huge collection of triples, however, without a common naming scheme (which is one of the defining characteristics of the Semantic Web, because it makes distributed contribution of contents easy). The query language of choice is therefore not SPARQL but keyword queries, maybe with a structured component. One notable such system is Siren [11]. It supports queries that correspond to star-shaped SPARQL queries, and predicates can be (approximately) matched by keywords. Since this is a very specific subclass of SPARQL queries, we do not include Siren in our evaluation in Section 5.

[5] https://virtuoso.openlinksw.com/

3 INDEXING

QLever has a knowledge-base index (Section 3.1) and a text index (Section 3.2). The knowledge-base index is designed such that the data needed for all the basic SCAN operations is stored contiguously and without any extra data in between. For the text index, we use redundancy to make sure that the data needed by the basic TEXT operations is stored contiguously. This is based on our previous work [5], but with a number of simple but effective improvements.

3.1 Knowledge-Base Index

The knowledge-base index is designed with the following goal: SCAN operations for query triples with either one or two variables should be as efficient as possible.

Like [17] and [20], we sort the triples (S = subject, P = predicate, O = object) in all possible ways and create six (SPO, SOP, PSO, POS, OSP, OPS) permutations. In practice, we have found that for typical semantic queries, two permutations (PSO and POS) suffice. With only these two permutations, one (only) loses the possibility to use variables for predicates. For QLever, the user can always decide to build 2 or all 6 permutations.

In the following, we use one permutation as our example: a PSO permutation for a *Film* predicate with actors as subjects and movies as objects. Figure 1 depicts example triples and the index lists we build for them. Other permutations are indexed accordingly.

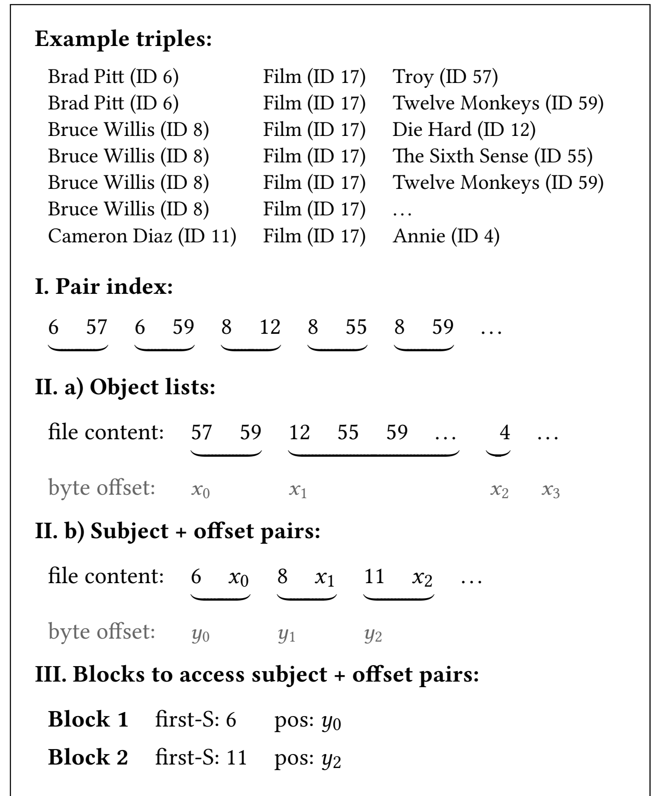


Figure 1: The components of our knowledge-base index, exemplified for a PSO permutation for a *Film* predicate.

To support a scan with two variables (i.e. for all triples with a given predicate), we simply keep all pairs of subject and object on disk: This pair index is displayed as part I in Figure 1. If we know where the list of pairs starts and the number of elements in it, we can directly read it into memory. This organization (pairs instead of all subjects followed by all objects) has turned out to be advantageous during query processing.

If a predicate is functional (only one object per subject) or relatively small (below a threshold, we currently use 10,000 as default value), then this is the only thing we index. In that simple case, a scan with only one variable (i.e. a scan for objects with given predicate and subject) is accomplished by finding the corresponding subject (with binary search) and taking the matching object(s) next to it. If we were looking for subjects, we would use the POS permutation instead.

For non-functional predicates the size of which exceeds our threshold, we create another index list that allows us to read matching objects without any overhead. Think of a predicate *has-instance* that holds type information for all entities in a large knowledge base and a scan like <*Person*> <*has-instance*> *?x*. Note that something like *?x* <*type*> <*Person*> would be equivalent and use a POS permutation instead. It is very important not to work through the entire *has-instance* list. Further, it is a huge benefit if a long list of person entity IDs can be read directly from disk without skipping over the ID of the type *Person* for every single one of them (as it would be the case in the pair index).

Therefore, we first index all object lists alone (part II.a in Figure 1). Separately, we store, for each subject, the byte offset of the start of its object list (part II.b). To quickly read a list of objects for a scan, we now only need to search for the correct entry in the list of (subject, byte offset) pairs and read between the byte offset and the byte offset of its successor. In the case of the scan for all persons, we read exactly the sorted list of person IDs from a continuous area in the index file and without any overhead.

However, there will still be predicates with many different subjects (and thus a long list of subject + offset pairs). We neither want to keep all of them in memory, nor read through all of them on disk. Thus, we split them into blocks. For each block, we then store the lowest subject ID in that block and the byte offset into the list of subject-offset pairs. This is depicted in part III of Figure 1. We keep this block information in memory (we also write it to the index but read it on startup). Now, the number of subjects + offset pairs we have to read for the scan only depends on the number of elements within a block and not on the number of subjects in the entire predicate. If we have a functional predicate that exceeds the size threshold, we also use the block information but let it point into the pair index. The object lists and subject-offset pairs are not necessary in that case.

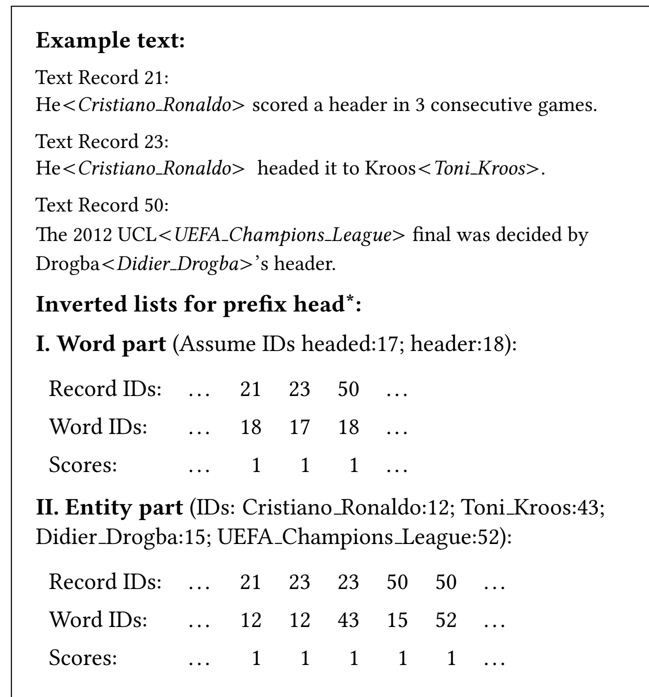
We want to remark that there is a trade-off between size and speed here. First of all, we could also answer all queries with the pair index, only. It would still be decently fast, just not ideal. Secondly, we could add compression: the repeated subjects in the pair index can be gap-encoded, and so can the objects lists and subject + offset pairs. We choose to not do this for a simple reason: For large collections, the size of the text corpus usually dominates the size of the knowledge base. If disk space consumption by the KB index should be problematic for some input, it would be possible to add compression without trouble and pay the price of slightly slower queries because of the time needed to decompress lists.

3.2 Text Index

Our text index is an improved version of the index presented in [5]. For a comparison with variants of a classic inverted index, we also refer to that paper. In particular, the index outperforms approaches where artificial words are inserted to represent entities and groups of entities (e.g., a special term for all entities of type *person*) and those where a classic inverted index is combined with a forward index to obtain co-occurring entities from matching text records.

Recall that we index text records that roughly correspond to sentences. The index items are tuples of text record ID, word ID, score (and optionally a position), sorted by text record ID. The word ID allows entity postings to be interleaved with the regular postings. This is where the main idea behind the index comes into play: To every inverted list, also add all co-occurring entities. This is basically a pre-computation for queries of the form "entities that co-occur with <*word*>" (without aggregation by entity).

For QLever, we also follow the main idea behind this index, but we split the list. Instead of interleaving word- and entity-postings, we keep two lists for each term (or prefix[6]). Thus, we end up with more, but shorter lists. Figure 2 illustrates these lists for a single prefix and a tiny example text excerpt.



Example text:

Text Record 21:
He<*Cristiano_Ronaldo*> scored a header in 3 consecutive games.

Text Record 23:
He<*Cristiano_Ronaldo*> headed it to Kroos<*Toni_Kroos*>.

Text Record 50:
The 2012 UCL<*UEFA_Champions_League*> final was decided by Drogba<*Didier_Drogba*>'s header.

Inverted lists for prefix head*:

I. Word part (Assume IDs headed:17; header:18):

Record IDs:	…	21	23	50	…
Word IDs:	…	18	17	18	…
Scores:	…	1	1	1	…

II. Entity part (IDs: Cristiano_Ronaldo:12; Toni_Kroos:43; Didier_Drogba:15; UEFA_Champions_League:52):

Record IDs:	…	21	23	23	50	50	…
Word IDs:	…	12	12	43	15	52	…
Scores:	…	1	1	1	1	1	…

Figure 2: The components of our text index, illustrated for an example text excerpt and the prefix "head*".

QLever makes a second improvement to the index: We only store word IDs when necessary: With prefix-search disabled or for a list

[6] Just like [5], we can optionally index prefixes instead of words and filter for exact words from those lists.

with only one word (e.g., for lists for short words, most stopwords, and all concrete entities which are treated just like words), the word ID list (in part I) becomes trivial. This is beneficial because of our first improvement. In the original index from [5], almost no list would be affected in that way, as there are (almost) always interleaved entities which we now keep separately.

The improvements have two main benefits: (1) An intelligent query execution can only read exactly what is needed; this is described in more detail in Section 4.4. (2) There is now zero overhead for classic full-text queries (no entities involved) because if we ignore the extra entity lists, we have a normal inverted index.

All lists are stored compressed. Just like in the index from [5], we gap-encode record IDs and frequency-encode word IDs and scores, and then use the Simple8b [2] compression algorithm on all of them.

3.3 Vocabulary

We do not store strings directly in either index but assign a numerical ID to all items from the KB and from the text, based on their lexicographical order. We also translate all values (float, integer and dates, in the input KB and from queries) into an internal representation where the lexicographical order corresponds to their actual order. We also support negative values. This enables efficient comparisons in Filter clauses by simply comparing the respective IDs.

We store the vocabulary partly in memory and partly on disk.[7] Items on disk start with a special character so that they come last in the lexicographic order. They are stored on disk as depicted in Figure 3.

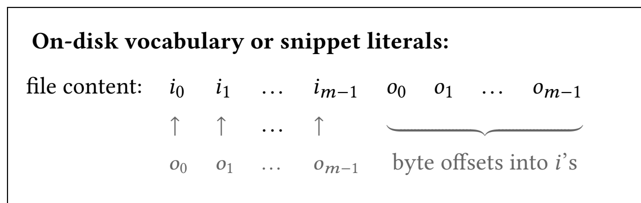


Figure 3: The first part of the file's contents are the m items i_j written without any separator. The second part are m offsets o_j into the first part of the file, and always mark the byte offset at the end of the respective element.

On startup, we read only the last l (= ID size, usually 8 or 4) bytes of the file. This gives us the location where o_0 is stored. With this information, we now have random access to the i'th item by just two seeks and reading $2 \cdot l + |i|$ bytes. We use the same data structure for providing result snippets for text queries (then with entire text records as items i_i).

4 QUERY PROCESSING

Our query processing has two parts: query planning and query execution. In the following, we describe the basic operations of QLever's query execution trees and their semantics (Section 4.1), our dynamic programming algorithm for query planning (Section 4.2), our heuristics for estimating result sizes and costs used during query planning (Section 4.3), and finally the algorithms to efficiently execute our basic operations (Section 4.4).

4.1 Basic Operations

The building blocks that comprise QLever's execution trees are from a fixed set of basic operations. Many of them are standard operations and their semantics should be self-explanatory, namely: Scan, Join, Sort, Distinct, Filter and Order_by. In the following, we describe the semantics of the less obvious operations.

Text_no_filter is a text operation that returns entities that co-occur with one or more words or concrete entities.

Input:	words or concrete entities
Output Columns:	record ID, score, entity ID, (…)
Options:	textlimit, #output-entities

The Textlimit limits the number of text records for each match (i.e. for each entity or for each combinations of entities depending on the #output-entities option) to include in the result. For simple entity-word co-occurrence, there are exactly 3 columns in the operation's result. When the SPARQL variable for the text record is connected to more than one other variable, then more than one entity column is required (which can be controlled through the #output-entities option). In that case, the text operation has to produce the cross product of co-occurring entities within each text record. This happens, for example, in the following query, which asks for persons who are friends with a scientist:

```
SELECT ?x WHERE {
  ?x <is-a> <Person> . ?y <is-a> <Scientist> .
  ?t ql:contains-entity ?x . ?t ql:contains-entity ?y .
  ?t ql:contains-word ”friend*”
}
```

Text_with_filter is a text operation similar to the one before, but with an additional sub-result as input. The operation has the same effect as a Join between the result of a Text_no_filter operation and the additional sub-result. However, it can be more efficient than computing the result of Text_no_filter and joining afterwards.

Input:	sub-result, filter column index in sub-result, words or concrete entities
Output Columns:	record ID, score, cols of sub-result, (…)
Options:	textlimit, #output-entities

The idea is that, e.g., for the query above, the list of scientists can be much smaller than the list of entities that co-occur with "friend*". Text_with_filter uses the list of scientists to filter the text postings as early as possible. This is described in Section 4.4. Figure 6 shows two example query plans for the same query that differ because of the kind of text operation they use.

Two_column_join is a Join where the joined sub-results have to match in two instead of one column.

[7] For Freebase, less than 1% of the literals use 50% of the space. We make use of that by externalizing long literals (>50 chars) to disk.

Input:	left sub-result, right sub-result,
	4 join column indices (2 left, 2 right)
Output Columns:	cols of left sub-result,
	cols of right sub-result w/o join columns

This operation is only relevant for ”cyclic” queries, e.g.:

SELECT ?a1 ?a2 ?f WHERE {
?a1 <Film_performance> ?f .
?a2 <Film_performance> ?f .
?a1 <Spouse> ?a2
}

A possible execution tree is to join the *Film_performance* lists on the column pertaining to the film (?f) and thus create all triples of actor+actor+movie that performed together in that film, and then use the *Spouse* list to filter it and only keep rows with pairs of actors that are also spouses. When we ”filter” by that *Spouse* list, we require two columns to match between the two sub-results and thus perform a Two_column_join.

4.2 Query Planning

For each SPARQL query, we first create a graph. Each triple pattern in the query corresponds to a node in that graph. There is an edge between nodes that share a variable. Figure 4 depicts the graph for the example query (*persons that are friends with a scientist*) from Section 4.1.

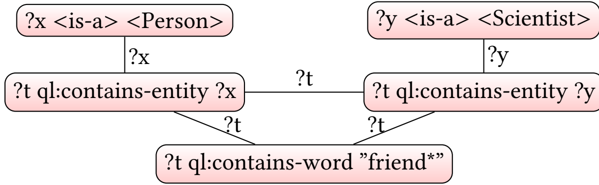


Figure 4: Graph for the first query from Section 4.1.

Text operations naturally form cliques (all triples are connected via the variable for the text record). We turn these cliques into a single node each, with the word part stored as payload. This is shown in Figure 5.

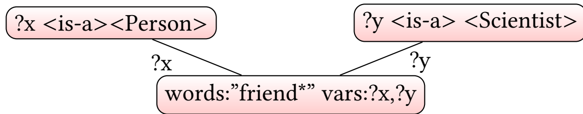


Figure 5: Text cliques collapsed for the graph from Figure 4.

We then build a query execution tree from this graph. Nodes of this tree are the basic operations discussed in Section 4.1. We rely on dynamic programming to find the optimal execution tree. This has already been studied for relational databases (see [16] for an overview) and has been adapted by SPARQL engines like RDF-3X.

Let n be the number of nodes in the graph. We then create a DP table with n rows where the k'th row contains all possible query execution trees for sub-queries of size k (= k nodes of the graph are included). We seed the first row with the n Scan (or Text_no_filter) operations pertaining to the nodes of the graph.

Then we create row after row by trying all possible merges. The k'th row is created by merging all valid combinations of rows i and j, such that $i + j = k$. A combination is valid if: (1) The trees do not overlap, i.e. no node is covered by both of them, and (2) there is an edge between one of their contained nodes in the query graph.

Whenever we merge two subtrees, a Join operation is created. Any subtree whose result is not yet sorted on the join column, is prepared by an extra Sort operation. There are two special cases: (1) If at least one subtree is a Text_no_filter operation, we create both possible plans: a normal Join and a Text_with_filter operation.[8] (2) If they are connected by more than one edge, we create a Two_column_join.

Before we return a row, we prune away execution trees that are certainly inferior to others: We only keep the tree with the lowest cost estimate for each group of equivalent trees. Trees are equivalent if they cover the same triples from the original query, cover the same Filter clauses from the original query, and their result tables are ordered by the same variable/column.

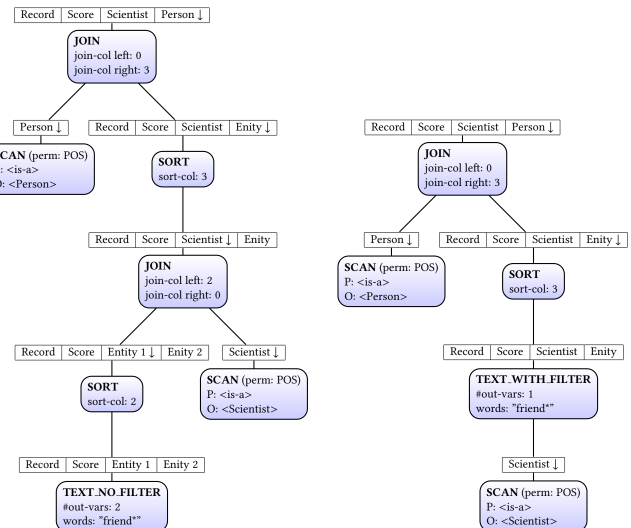


Figure 6: Two (out of many possible) example execution trees for the query from Figure 5. The right one is smaller, because of the complex Text_with_filter operation.

After each row, we apply all Filter operations possible (i.e. all variables from the filter are covered somewhere in the query). For the next round, each remaining candidate is considered in two variants: with all possible filters applied and with none of them applied (but not with all subsets of filters). The exception is the last row, where all Filters have to be taken. Modifiers like Order_by or Distinct are applied in the end. Finally, the tree with the lowest cost estimate is used. Figure 6 shows two of many possible execution trees that are created for the example query and graphs from above.

[8] When a Text_with_filter operation is created, one subtree is kept as a child and the Text_no_filter operation is removed / included in the operation.

4.3 Cost, Size and Multiplicity Estimates

To decide which execution tree to prefer, we need to estimate their costs. If we know the sizes of all (sub)-results, cost estimates are straightforward: We count the elements touched (usually just the size of in- and output), or use a simple function like $n \cdot log(n)$ for a SORT and account for differences between hashmap and array access.

The interesting part is getting estimates for the size s of a sub-result and the number of distinct elements d_i within each column (or their multiplicity $m_i = s/d_i$). These values are most interesting for SCAN, JOIN and TEXT operations. In the following, we describe how they are computed. For SCANS this is rather easy. For JOIN and TEXT operations, the computation is more complex, but getting these estimates right is crucial for finding good execution trees.

SCAN: For SCAN operations with two variables, we know their size from what we keep in memory for each pair index (see Figure 1). We also compute both multiplicities m_0, m_1 when we construct the index and keep their logarithms in memory (1 byte each). For SCAN operations with only one variable, we simply execute the SCAN before the query planning. In that case, we also know that $d_0 = s$, as there should not be any duplicates.

JOIN: For JOIN operations we compute the size as

$$s := \alpha \cdot m_a \cdot m_b \cdot \min(d_a, d_b)$$

where m_a and m_b are the multiplicities of the join columns in their respective sub-results (i.e. in the two tables used as input to the join), d_a and d_b are the numbers of distinct entities in them, and α is a correction factor (because not all elements from the join columns will match their counterparts). We also have to calculate all d_i in the result of the join. Therefore, we regard *orig*, the input of the join from which the result column i originates. Let s_{orig} be the size of that input and $d_{join,orig}$ be the number of distinct elements in its join column. Let $d_{\text{i,orig}}$ be the number of distinct elements in the column that becomes column i after the join. Further, let $d_{\text{join,other}}$ be the number of distinct elements in the join column of the other input/operand of the join. We then adjust the size of *orig* to the portion that can at most match in the join:

$$s'_{\text{orig}} := s_{\text{orig}} \cdot \alpha \cdot \min(d_{\text{join,orig}}, d_{\text{join, other}})/d_{\text{join, orig}}$$

With this we can estimate the new number of distinct elements as:

$$d_i := \min(d_{\text{i, orig}}, s'_{\text{orig}})$$

TEXT: For TEXT_NO_FILTER, we first describe the simple case where TEXTLIMIT is 1 and only one co-occurring entity shall be returned in each result row: We estimate

$$s := \max(1, l/100)$$

where l is the length of the entity part (see Figure 2) of the smallest involved index list In this simple case, all multiplicities are 1. If TEXTLIMIT $t > 1$ or with more than one entity to co-occur within a record, this does not remain true. Let s_0 be the estimated number of entities that co-occur with the text part, i.e. the s from the simple case. Let o be the desired number of output entities to co-occur within each record. Let e be the average number of occurrences per entity across the whole collection. We account for the TEXTLIMIT as $s_t := s_0 \cdot \min(t, e)$ and then for multiple output entities as $s_{final} = \text{pow}(s_t, o)$. All column multiplicities are then $m_i := \text{pow}(s_t, (o-1))$.

For TEXT_WITH_FILTER, we simply compute s, d_i and m_i as we would for the equivalent combination of TEXT_NO_FILTER and extra JOIN operation.

4.4 Query execution

We compute results for our execution trees in a bottom up fashion and employ an early materialization strategy (see [1] for an overview of different strategies). Each operation in our query processing constructs a result table with columns for all involved variables (see also Figure 6). We do not do any pipelining. This has two benefits: (1) the simplicity allows us to create very efficient routines for our operations; (2) we can cache and reuse all sub-results within and across queries.

Most of our basic operations from Section 4.1 are implemented in the straightforward way. All JOIN operations are realized as merge joins using the straightforward linear-time "zipper" algorithm. If one side is much smaller than the other, we use binary search to advance in the larger list. In the following we describe operations with non-obvious algorithms.

For **TEXT_NO_FILTER** we compute tuples (*record ID*, *score*, *entity ID*). Such a tuple means that the entity co-occurs with all words of the text operation in k records, where k is the *score*.[9] The *record ID* is from one of these k records. How many of the k records appear in the result is controlled by the TEXTLIMIT operator.

For the word with the smallest index list[10], we first obtain the precomputed list of co-occurring entities (the "Entity Part" in Figure 2). For each other word, we obtain the precomputed inverted list of occurrences in the text records (the "Word Part" in Figure 2). This is a significant improvement over the query processing in [5], where all the information is read for all lists. We then intersect (using k-way "zipper") these lists on record ID. For each pair of record and entity, we aggregate (sum up) the scores. This gives us parallel lists of *record ID*, *score*, *entity ID*, sorted by record ID.

We then aggregate by entity and, for each entity, keep the t records with the largest score, where t is given by the TEXTLIMIT modifier (default: $t = 1$). We achieve this via a single scan over the records and a hash map, which for each entity maintains a sorted set of at most t (record, score) pairs, containing the records with the largest scores seen so far. We count the number of all matching records for each entity (or combination of entities) and use this as the score in the final tuple. In some queries, the operation may need to produce more than one co-occurring entity (recall the example query in Section 4.1). In that case, we build the cross product of entities within each text record and use the hash map above with entity tuples (instead of single entities) as keys.

For a **TEXT_WITH_FILTER** operation, we store the sub-result in a hash set. We use it to filter postings from the entity list, that we read from our index (see above), before we intersect with other word lists. When filtering, we keep all postings from all records that contain an entity from the filter. After the intersection, we are often left with a lot fewer postings that we have to aggregate subsequently.

Another interesting case are JOINS with triples that consist of three variables. Since our index is optimized for SCAN operations

[9] More sophisticated scoring schemes are possible, too.

[10] The length of an index list for a word is the total number of occurrences of all words with the same prefix, see Section 3.2.

with one or two variables, we have to follow a different strategy here. We avoid a full index SCAN for possibly billions of triples. Instead we perform a SCAN for each distinct entity that is to be joined with the full index. Our query planning ensures that we organize execution trees so that there are few of these distinct entities.

5 EVALUATION

We evaluate QLever by comparing it against three systems that each are, to our best knowledge, the most efficient representatives of their kind: RDF-3X, a pure SPARQL engine with a purpose-built index and query processing; Virtuoso, a commercial SPARQL engine, built on top of a relational database, with its own text-search extension; and Broccoli, an engine for combined search on a knowledge base and text, which supports a subset of SPARQL.

For **RDF-3X**, we use the latest version 0.3.8 and add an explicit *contains* predicate with triples such as *<record:123> <contains> <word:walk>* and *<record:123> <contains> <Neil_Armstrong>*. This is enough to answer all queries that do not involve prefix search. We have also tried using two predicates, *contains-word* and *contains-entity*. For most queries the effects were small, sometimes leading to slightly faster query times, sometimes to slightly slower query times. However, there was a single query where using two predicates caused it to take over 3000 seconds (presumably due to a bug or weakness in the query planning), thus unrealistically increasing average times. Therefore, we report results for runs with a single *contains* relations for RDF-3X.

For **Virtuoso**, we use the latest stable release, version 7.2.4.2, and configure it to use the largest recommended amount and size of buffers (we have tried using even more, without noticeable positive effects). We also add triples of an artificial *contains* predicate between text records and entities. The text records themselves are each connected to a text literal with the record's contents, such as *<record:123> <contains> <Neil_Armstrong>* and *<record:123> <content> "He walked on the moon"*. We use Virtuoso's full-text index for these literals and use its special *bif:contains* predicate to search them. We also considered other setups (including less expressive ones without explicit text-record entities) which we describe as variants in Section 5.4.

Broccoli natively supports queries with text search similar to QLever. However, it only searches for lists of entities and never for entire tuples. Queries have to be tree-shaped. This limits the number of queries in our evaluation that can be answered by Broccoli.

We conducted all experiments on a server with a Intel Xeon CPU E5-1630 v4 @ 3.70GHz and 256GB RAM. Before the run for each approach, we explicitly cleared the disk cache and then ran our entire query set within one go. In Section 5.4, we also report results for runs with warm caches and find a speedup factor of roughly two across all systems.

5.1 Datasets

We evaluate the systems described above on two datasets.

FreebaseEasy+Wikipedia: This dataset consist of the text of all Wikipedia articles from July 2016 linked to the FreebaseEasy knowledge base [4]. FreebaseEasy is a derivation from Freebase [9] with human-readable identifiers, a reified schema (without mediator objects), and containing all the Freebase triples with actual "knowledge" but discarding many "technical" triples and non-English literals. The example queries used throughout this paper use entity and predicate names from FreebaseEasy. The version used in our experiments has 362 million triples, the Wikipedia text corpus has 3.8 billion word and 494 million entity occurrences. This dataset is similar to the Wikipedia LOD dataset used in many of the INEX benchmarks (see Table 2.3 from [6]), but with about seven times more triples and a more recent version of Wikipedia.

Freebase+ClueWeb: This dataset is based on FACC [13], which is a combination of Freebase [9] and ClueWeb12 [10]. We omitted stopwords (which have no effect on query times when they are not used in queries) and limit ourselves to annotations within sentences (the FACC corpus also contains annotations within titles and tables but these are not useful for our kind of search). For this dataset, we use the less readable original Freebase dataset, because the FACC corpus links Freebase's machine IDs to their occurrences in the text. This also gives us a larger set of triples to index. The resulting dataset has a knowledge base with 3.1 billion triples and a text corpus with 23.4 billion word and 3.3 billion entity occurrences. These numbers are roughly ten times larger than for the FreebaseEasy+Wikipedia dataset.

5.2 Queries

We distinguish the following 12 sets of queries and report average query times for each of them. This explicitly shows which kinds of queries cannot be answered by one of the systems at all, and which kinds of queries are hard to answer for which system.

One Scan: 10 queries that can be answered with a single scan.
One Join: 10 queries that can be answered with a single join between the result of two scans.
Easy SPARQL: 10 pure SPARQL queries with small result sizes.
Complex SPARQL: 10 SPARQL queries that involve several joins, either star-shaped, paths, or mixed.
Values + Filter: 10 SPARQL queries that makes use of values (integers, floats or dates) and FILTER operations on them that compare against fixed values or each other.
Only text: 10 queries that do not involve a KB part. One or multiple words and prefixes are used to search for matching records (5 queries) or co-occurring entities (5 queries).
Is-a + Word: 10 queries for entities of a given type that co-occur with a given term.
Is-a + Prefix: 10 queries for entities of a given type that co-occur with a given prefix.
SemSearch W: 49 queries from the SemSearch'10 challenge, converted to SPARQL+Text queries without using word prefix search.
SemSearch P: 10 queries from the SemSearch'10 challenge, converted to SPARQL+Text queries using word prefix search.
Complex Mixed: 10 queries that mix several knowledge-base and text triple patterns, sometimes nested; no prefix search involved.
Very Large Text: 10 queries that return or involve a very large number of matches from the text; most use word prefixes.

The queries within most of these sets were chosen by hand with the goal to create queries with a sensible narrative. For example, a query from the *Very Large Text* set is for soccer players who played somewhere in Europe and also somewhere in the US. The

upside is, that those queries are more realistic than automatically generated ones, especially w.r.t the selectivity of combinations with subqueries. The downside is, obviously, that they are hand-picked. Therefore we also incorporated the queries from the SemSearch Challenge's [8, 14] query set by translating them to SPARQL+Text queries (in the straightforward way). These queries are based on real user queries from search engine log files and can be answered very well using our corpus. We omitted three queries where the desired results were not contained in Freebase. For example, Freebase does not contain the relevant entities for the query *axioms of set theory*. The remaining queries can mostly be expressed by specifying a type and several words to co-occur. Some require entity-entity co-occurrence, some can be answered using only the knowledge base, and 10 of them can be expressed much better using a prefix. Since word prefix search is not possible in RDF-3X and significantly slower in Virtuoso, we put these queries into their own categories.

For text queries, we have not included the text records (neither the ID nor the text) in the SELECT clause of the query and we use a DISTINCT modifier for the remaining variables. This is important for fairness, because otherwise competitors would have to produce much larger output for some queries than Broccoli and QLever.

For the Freebase+ClueWeb dataset we use the same queries, but adapt them to Freebase. Finding the corresponding predicates, types, and entity IDs is not always easy and thus manual translation of queries is a lot of work. Therefore, we only translate the most interesting query sets, in particular the ones that also involve text; see Table 2 below.

5.3 Results

FreebaseEasy+Wikipedia: Table 1 lists the average query times for each of the categories as well as the size of the index on disk and the amount of memory that was used (by the process; while runs all started with an empty disk cache, it may have been used during the run). QLever is fastest across all categories and produces the fastest result for 89% of all individual queries.

The results are not surprising for SPARQL+Text queries, for which QLever (and to some extent also Broccoli) was explicitly designed. However, QLever also beats the competition on pure SPARQL queries. Most notably, the difference is large for the *Complex SPARQL* set. The price payed for this efficiency is best reflected in the index size without text. We deliberately add redundancy in our knowledge-base index (see Section 3.1) for the sake of fast query times. The reason for this choice is reflected in the total index size: with a large text corpus, the size of the knowledge base becomes less and less relevant, but effective compression of the text index is much more important.

Freebase+ClueWeb: Table 2 reports query times of QLever for the five hardest query categories. We do not report numbers for the other systems, because they failed to index this large dataset in reasonable time on our available hardware (256 GB RAM and more than enough disk space). For Virtuoso, we aborted the loading process after two weeks.

[11]The size of the index files needed to answer the queries from this evaluation is actually only 52 GB. Not all permutations of the KB-index are necessary for the queries, but virtuoso and RDF-3X build them as well and, unlike QLever, do not keep them in separate files.

[12]20 GB for the permutations that are really needed.

[13]All systems were set up to use as much memory as ideally useful to them. All of them are able to answer the queries with less memory used.

Table 1: Average query times for queries from 12 categories (Section 5.2) on FreebaseEasy+Wikipedia (Section 5.1).

	RDF-3X	Virtuoso	Broccoli	QLever
One Scan	584 ms	1815 ms	162 ms	47 ms
One Join	743 ms	2738 ms	117 ms	41 ms
Easy SPARQL	98 ms	337 ms	-	74 ms
Complex SPARQL	3349 ms	14.2 s	-	262 ms
Values + Filter	623 ms	430 ms	-	59 ms
Only Text	10.7 s	15.0 s	427 ms	191 ms
Is-a + Word	1776 ms	941 ms	178 ms	78 ms
Is-a + Prefix	-	20.5 s	310 ms	118 ms
SemSearch W	1063 ms	766 ms	196 ms	74 ms
SemSearch P	-	107.8 s	273 ms	125 ms
Complex Mixed	5876 ms	13.6 s	-	208 ms
Very Large Text	-	3673 s	632 ms	605 ms
Index Size	138 GB	124 GB	39 GB	73 GB[11]
Index w/o Text	17 GB	9 GB	8 GB	49 GB[12]
Memory Used[13]	30 GB	45 GB	10 GB	7 GB

For the sake of practical relevance, we considered two variants of QLever, or rather of the queries. For the first variant (QLever), we use the same queries as for FreebaseEasy+Wikipedia, which for Freebase yields only IDs. For the second variant (QLever+N), we use enhanced queries that return human-readable names. This is achieved by adding a triple *?x fb:type.object.name.en ?xn* for each result variable *?x* (and replacing *?x* by *?xn* in the SELECT clause), where the predicate is the subset of Freebase's huge *type.object.name* predicate restricted to English.

Table 2: Average query times for QLever for queries from the 5 hardest categories (Section 5.2) on Freebase+ClueWeb (Section 5.1). For QLever+N, queries have been augmented such that the result does not just contain the Freebase IDs but also the Freebase names.

	cold cache		warm cache	
	QLever	QLever+N	QLever	QLever+N
Only Text	1279 ms	1382 ms	840 ms	881 ms
SemSearch W	390 ms	479 ms	214 ms	262 ms
SemSearch P	613 ms	755 ms	339 ms	376 ms
Complex Mixed	1021 ms	1273 ms	603 ms	714 ms
Very Large Text	2245 ms	2289 ms	1849 ms	1885 ms

We can see that this very large dataset (almost 10 times larger than FreebaseEasy+Wikipedia), does not cause any problems for QLever. We are confident that an increase in size by another order

of magnitude would be no problem either. However, we are not aware of a knowledge base linked to a text corpus of that dimension.

5.4 Variants

To verify the robustness of our results, we considered a few variants of the setup described above, concerning caching and the realization of searching the text corpus.

Warm caches: Table 3 reports the same values as Table 1 but with warm disk caches. That is, all parts of the index files relevant for the benchmark are cached in main memory. The application caches (that is, whatever caching the systems use internally) are still empty. All systems perform roughly twice as fast compared to the runs with cold caches.

Table 3: Repetition of the experiments from Table 1 with warm disk cache.

	RDF-3X	Virtuoso	Broccoli	QLever
One Scan	534 ms	1254 ms	118 ms	30 ms
One Join	711 ms	3036 ms	47 ms	13 ms
Easy SPARQL	45 ms	170 ms	-	16 ms
Complex SPARQL	2475 ms	4505 ms	-	125 ms
Values + Filter	532 ms	465 ms	-	30 ms
Only Text	3638 ms	10.3 s	304 ms	82 ms
Is-a + Word	1715 ms	429 ms	85 ms	30 ms
Is-a + Prefix	-	14.6 s	145 ms	58 ms
SemSearch W	991 ms	397 ms	112 ms	28 ms
SemSearch P	-	107 s	175 ms	43 ms
Complex Mixed	2775 ms	5127 ms	-	68 ms
Very Large Text	-	1799 s	533 ms	439 ms

Application caches and query order: The internal caching mechanisms of the various systems are hard to compare. While Virtuoso caches only parts of the queries, Broccoli and QLever can cache entire queries, so that a repetition of a query is instant. The order of queries within a run does have an effect: All approaches take longer when they access large lists for the first time, e.g., when they scan for all persons or the first time Virtuoso or RDF-3X access the data for the *contains* predicate. Therefore, we compared random permutations of our queries. This lead to some distortion across query sets but the overall average over all categories never changed significantly. For the tables above, we made sure that all approaches were fed the queries in the same order.

Virtuoso variants: We also evaluated a variant of Virtuoso without *bif:contains*, just like we did for RDF-3X. The results were similar but slightly worse (14% slower overall) than for RDF-3X. We tried another variant of Virtuoso where, instead of fully simulating *ql:contains*, we directly connected entities with literals for the text records (one triple per record-contains-entity) and search them via *bif:contains*. This was faster than the approach reported in Table 1 but still within the same order of magnitude (hence much slower than QLever) and without the possibility to express entity-entity co-occurrences so that not all the queries could be answered.

6 CONCLUSIONS

We have presented QLever, a search engine for the efficient processing of SPARQL+Text queries on a text corpus linked to a knowledge base. For queries using both the SPARQL and the Text part, QLever outperforms existing engines by a large margin, and it is also better for pure SPARQL queries. On a single machine, QLever works on datasets as large as Freebase+ClueWeb (23.4 billion words, 3.1 billion triples), which other engines failed to process in a reasonable time on a single machine.

QLever could and should be developed further in several directions. So far, incremental index updates (INSERT operations) are not supported. Caching plays an important role already (by reusing results for subtrees of the query), but more sophisticated schemes could boost performance further in practice. A convenient user interface (maybe inspired by [3]) would be important to ease the process of query construction and to be able to explore a given dataset.

REFERENCES

[1] D. J. Abadi, D. S. Marcus., D. J. DeWitt, and S. R. Madden. 2007. Materialization strategies in a column-oriented DBMS. In *ICDE*. IEEE, 466–475.
[2] V. N. Anh and A. Moffat. 2010. Index compression using 64-bit words. *Softw., Pract. Exper.* 40, 2 (2010), 131–147.
[3] H. Bast, F. Bäurle, B. Buchhold, and E. Haussmann. 2012. Broccoli: semantic full-text search at your fingertips. *CoRR* abs/1207.2615 (2012).
[4] H. Bast, F. Bäurle, B. Buchhold, and E. Haußmann. 2014. Easy access to the Freebase dataset. In *WWW*. 95–98.
[5] H. Bast and B. Buchhold. 2013. An index for efficient semantic full-text search. In *CIKM*. 369–378.
[6] H. Bast, B. Buchhold, and E. Haussmann. 2016. Semantic Search on Text and Knowledge Bases. *Foundations and Trends in Information Retrieval* 10, 2-3 (2016), 119–271.
[7] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. 2007. ESTER: efficient search on text, entities, and relations. In *SIGIR*. 671–678.
[8] R. Blanco, H. Halpin, D. M. Herzig, P. Mika, J. Pound, H. S. Thompson, and D. T. Tran. 2011. Entity search evaluation over structured web data. In *SIGIR-EOS*.
[9] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*. 1247–1250.
[10] ClueWeb. 2012. (2012). The Lemur Projekt http://lemurproject.org/clueweb12.
[11] R. Delbru, S. Campinas, and G. Tummarello. 2012. Searching web data: An entity retrieval and high-performance indexing model. *J. Web Sem.* 10 (2012), 33–58.
[12] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. Meral Özsoyoglu. 2009. A complete translation from SPARQL into efficient SQL. In *IDEAS*. 31–42.
[13] E. Gabrilovich, M. Ringgaard, and A. Subramanya. 2013. (2013). FACC1: Freebase annotation of ClueWeb corpora, Version 1 Release date 2013-06-26, Format version 1, Correction level 0, http://lemurproject.org/clueweb12/FACC1.
[14] H. Halpin, D. Herzig, P. Mika, R. Blanco, J. Pound, H. Thompson, and D. T. Tran. 2010. Evaluating ad-hoc object retrieval. In *IWEST*.
[15] M. Joshi, U. Sawant, and S. Chakrabarti. 2014. Knowledge Graph and Corpus Driven Segmentation and Answer Inference for Telegraphic Entity-seeking Queries. In *EMNLP*. ACL, 1104–1114.
[16] G. Moerkotte and T. Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*. 930–941.
[17] T. Neumann and G. Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113.
[18] B. Popov, A. Kiryakov, D. Ognyanoff, D. Manov, and A. Kirilov. 2004. KIM - a semantic platform for information extraction and retrieval. *Natural Language Engineering* 10, 3-4 (2004), 375–392.
[19] V. Tablan, K. Bontcheva, I. Roberts, and H. Cunningham. 2015. Mímir: An open-source semantic search framework for interactive information seeking and discovery. *J. Web Sem.* 30 (2015), 52–68.
[20] C. Weiss, P. Karras, and A. Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019.
[21] M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. 2011. dipLODocus[RDF] - Short and Long-Tail RDF Analytics for Massive Webs of Data. In *ISWC*. 778–793.