

Efficient Semantic Search on Very Large Data

Dissertation zur Erlangung des Doktorgrades
der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg

vorgelegt von
Björn Buchhold

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik
2017

Abstract

This thesis is about efficient semantic search on very large data. In particular, we study search on combined data, which consists of a knowledge base and a text corpus. Entities from the knowledge base are linked to their occurrences within the text, so that queries can be answered with the help of all structured information from the knowledge base and all unstructured information available in the text corpus. This deep integration through entity occurrences and the possibility to fully take advantage from it within queries distinguishes our work from previous approaches. We introduce Broccoli, a novel kind of search engine that can be set up for any knowledge base and text corpus that are linked through recognized entity occurrences and that provides a convenient way of searching the data through its user interface.

Further, we present QLever, a query engine for efficient combined search on a knowledge base and text. We call QLever’s query language SPARQL+Text. This language extends SPARQL, the de-facto standard for knowledge-base queries, by two special predicates, *ql:contains-word* and *ql:contains-entity*. In this way, we provide a standard interface to the search capabilities of Broccoli. This ensures that our work can conveniently be used as a component in other systems, e.g., for question answering. In terms of expressiveness, the entire potential of the Broccoli search engine is retained and even extended so that we provide full SPARQL support. Efficiency is a primary concern of our work. Compared to state-of-the-art SPARQL engines, QLever is often faster on classic SPARQL queries, and several orders of magnitude faster on the SPARQL+Text queries it was specifically made for.

In this document, we summarize our contributions on, and the evaluation of, the search paradigm itself, efficient indexing and query processing, as well as machine-learned relevance scores that improve ranking of search results from the knowledge base. In the process, we present the ten publications that comprise this thesis. All of this research is directly applied in the creation of our software systems, Broccoli and QLever. By doing so, we make sure that our work yields open-source software that is accessible by the public and that all of our experiments are easily reproducible.

Zusammenfassung

Diese Dissertation beschäftigt sich mit effizienter semantischer Suche auf sehr großen Daten. Im Speziellen betrachten wir Suche auf kombinierten Daten, die einerseits aus einer Wissensdatenbank (Knowledge Base), andererseits aus einem Textkorpus, bestehen.

Für unsere Zwecke ist eine Knowledge Base eine Sammlung von Tripeln, bestehend aus Subjekt, Prädikat und Objekt. Jedes solche Tripel beschreibt einen Fakt. So drücken beispielsweise die beiden Tripel $\langle \textit{Pantheon} \rangle \langle \textit{is-a} \rangle \langle \textit{Building} \rangle$ und $\langle \textit{Pantheon} \rangle \langle \textit{located-in} \rangle \langle \textit{Europe} \rangle$ aus, dass das Pantheon ein Gebäude in Europa ist. Während Vorverarbeitungsschritten werden die Entitäten (in der Regel Subjekte und Objekte, theoretisch sind auch Prädikate möglich) dieser Knowledge Base im Textkorpus durch Entity Recognition erkannt und disambiguiert. D.h. jedes Vorkommen einer Entität wird mit ihrer eindeutigen ID aus der Knowledge Base annotiert. Zum Beispiel im folgenden Satz: *The Augustan $\langle \textit{Augustan Age} \rangle$ Pantheon $\langle \textit{Pantheon} \rangle$ was destroyed in a fire.*

Unter Berücksichtigung sowohl der Tripel der Knowledge Base als auch des Textes lässt sich das Pantheon als eine der Antworten auf eine Anfrage nach Gebäuden in Europa, die einmal in einem Feuer zerstört wurde, finden. Mit jeweils nur eine dieser Datenquellen wäre dies nicht möglich gewesen: Die Knowledge Base enthält nicht die eher spezifische Information, dass das Pantheon einmal in einem Feuer zerstört wurde, und aus dem Textabschnitt geht nicht hervor, dass es sich bei “Pantheon” um das Gebäude handelt und dass dieses in Europa steht. Anfragen dieser Art unterscheiden sich von der klassischen Suche nach relevanten Dokumenten, werden aber jederzeit vielfach von Nutzern großer Suchmaschinen gestellt. Gerade im Rahmen von Recherchen, egal ob durch Historiker, Anwälte, Recruiter, Journalisten, etc., tauchen Anfragen dieser Art immer wieder auf.

In dieser Dissertation präsentieren wir Broccoli, unsere neuartige Suchmaschine, die für beliebige Kombinationen aus Knowledge Base und Textkorpus aufgesetzt werden kann und Nutzern durch ihr User Interface eine praktische Möglichkeit bietet, solche kombinierten Daten zu durchsuchen. Somit können semantische Anfragen unter Betrachtung aller strukturierter Informationen aus der Knowledge Base und aller unstrukturierter Informationen aus dem Textkorpus beantwortet werden. Die tiefgreifende Verbindung über Vorkommen der Entitäten im Text und die Möglichkeit in Anfragen Informationen aus Text und Knowledge Base beliebig zu schachteln unterscheidet unsere Arbeit dabei von bisherigen Ansätzen.

Des Weiteren präsentieren wir QLever, ein System, das auf effiziente, kombinierte Suche auf Text und Knowledge Bases spezialisiert ist. Wir nennen unsere Anfragesprache SPARQL+Text. Diese Sprache erweitert SPARQL, den De-facto-Standard für Anfragen auf Knowledge Bases, um zwei künstliche Prädikate, *ql:contains-word* und *ql:contains-entity*. Auf diesem Weg stellen wir eine standardisierte Schnittstelle zu den weitreichenden Einsatzmöglichkeiten von Broccoli bereit. Somit kann unsere Forschung als Komponente in anderen Systemen, etwa für Question Answering, genutzt werden. Ein Schwerpunkt unserer Arbeit liegt dabei auf Effizienz: Verglichen mit state-of-the-art Systemen für SPARQL-Suche liefert QLever im Schnitt schnellere Antwortzeiten für klassische SPARQL-Anfragen und ist auf Anfragen, die SPARQL+Text nutzen und für die es speziell entwickelt wurde, um Größenordnungen überlegen.

Dieses Dokument ist wie folgt aufgebaut: In Kapitel 1 stellen wir unser Suchparadigma und die wichtigsten Beiträge unserer Arbeit genauer vor. In Kapitel 2 betrachten wir verwandte Ansätze aus der Forschung und von kommerziellen Suchmaschinen. Kapitel 3 listet die zehn Publikationen auf, die diese kumulative Dissertation ausmachen und benennen jeweils die Art der Publikation, ihren Inhalt und in welchem Umfang die einzelnen Autoren an der Arbeit beteiligt waren. In Kapitel 4 beschreiben wir die Forschungsfelder, denen unsere Publikationen zugeordnet werden können: unsere Arbeit am neuartigen Suchparadigma selbst (Kapitel 4.1), an effizienten Datenstrukturen für die Indizierung der Daten sowie an Algorithmen zur Beantwortung von Anfragen (Kapitel 4.2) und zu maschinell gelernten *Relevance Scores* für Tripel der Knowledge Base, die es uns erlauben Suchresultate sinnvoll zu ranken (Kapitel 4.3). Für jedes dieser Forschungsfelder formulieren wir eine prägnante Problembeschreibung, ordnen unsere Arbeiten neben verwandten Ansätzen aus der Forschung ein, stellen unseren konkreten Lösungsansatz vor und präsentieren die Ergebnisse unserer experimentellen Evaluation. Im Anschluss an die Beschreibung der Forschungsfelder fassen wir kurz unseren umfassenden Übersichtsartikel zum Thema “Semantic Search on Text and Knowledge Bases” zusammen (Kapitel 4.4). Die Arbeit schließt mit einem Fazit in Kapitel 5.

Unsere gesamte Forschung findet direkt in unseren Systemen, Broccoli und QLever, Anwendung. Dadurch stellen wir sicher, dass unsere Arbeit als Open-Source-Software der Öffentlichkeit zugänglich ist und dass all unsere Experimente reproduzierbar sind.

Acknowledgements

Foremost, I want to thank my supervisor Hannah Bast who provided invaluable guidance and support whenever I needed it. The way I now approach a problem, think about it, and iterate on possible solutions has been decisively influenced by you. I think one could say, that you made me the computer scientist I am today and I hope that you can take this as a compliment and not as an accusation.

I am grateful towards all my colleagues and fellow PhD students, Elmar Haußmann, Florian Bäurle, Claudius Korzen, Patrick Brosi, Niklas Schnelle and Markus Näther. I sincerely enjoyed our discussions over lunch – about computer science and also just about everything else. I am deeply grateful for your proofreading of parts of this thesis and of the included publications. Even more importantly, I want to thank you for making my daily life as a PhD student enjoyable: Thanks for playing football with me every Tuesday and for evenings out.

With all my heart, I want to thank my beloved girlfriend Linda, who supported me throughout the years. I cannot imagine having accomplished this without you and I hope there will be many years in the future during which I can try to return the favor. Finally, I want to thank my family. I cannot express how grateful I am for all the support during the last years and my entire life.

Contents

Abstract	I
Zusammenfassung	II
Acknowledgements	IV
1 Introduction	1
1.1 Contributions	5
2 Related Work	7
2.1 Search with (Semi-) Structured Queries	7
2.2 SPARQL Engines	8
2.3 Semantic Web Search	11
2.4 Commercial Search Engines and KBs	12
3 Publications	15
3.1 Peer-Reviewed Publications	15
3.2 Other Publications	19
4 Research Topics	20
4.1 Semantic Full-Text Search with Broccoli	20
4.2 Efficient Indexing and Query Processing	32
4.3 Relevance Scores for Knowledge-Bases Triples	46
4.4 Survey: Semantic Search on Text and Knowledge Bases	52
5 Conclusion	55
6 References	57

1 Introduction

This thesis describes a novel search paradigm for search in text and knowledge bases and the various components and research topics that empower a system for efficient and effective querying of such combined data. There is an emphasis on efficiency, but indirectly this also leads to improved effectiveness: the larger the data we can search whilst maintaining convenient response times, the better the results.

Assume someone is looking for *buildings in Europe that once were destroyed in a fire*. This is called an entity query, a query for things. Entity queries are different from classic document queries, where a search engine finds documents about the query keywords. However, a large part of search engine queries are actually about entities. Queries like the example above are typical for people doing research on a topic. For historians, journalists, recruiters, and many more, it is very common to seek information about things or people that satisfy certain criteria. In a study on a large query log from a commercial web-search, Pound, Mika, and Zaragoza, (2010) have found that for nearly 60% of queries, entities were the primary intend of the query and that more than 12% were of the same kind as our example.¹ Additionally, such queries can contribute to question answering systems as an important component.

We answer these queries using data that consists of a knowledge base (KB) on the one hand, and of a text corpus on the other hand. Modern knowledge bases are collections of high-precision statements (sometimes also denoted as *facts*) and can be queried with exact semantics. Recent and very specific information, however, is usually not included in them but only available as free text. Thus, we rely on both sources and leverage the strengths of either to overcome the weaknesses of the other. In our publication (Bast et al., 2012a), we have argued for the importance of this search paradigm.

To provide an intuition of how our approach works, let us now look at how the Pantheon in Rome, a reasonable match for our example query, can be returned as one of the results. Text, which states that it was once destroyed by fire, is widely available. Some passages are obvious, for example in the Wikipedia article about the building itself:

Wikipedia article: Pantheon

[...] The Augustan Pantheon was destroyed along with other buildings in a huge fire in 80. [...]

However, one does not solely want to rely on finding the information in articles specifically about the entity. There is usually much more useful information available in other documents as well. For example, even within Wikipedia the following article about the Roman emperor Hadrian also contains the fact we are looking for:

¹To reach the very high figure of 60%, they include queries for a particular entity (e.g., *pantheon*) that make up 40% of queries and are arguably very similar to classic document queries.

Wikipedia article: Hadrian

[...] In Rome, the Pantheon, originally built by Agrippa but destroyed by fire in 80, was rebuilt under Hadrian in the domed form it retains to this day. [...]

In both text snippets we have underlined mentions of entities. Ideally, we want to make use of all, or at least most, such entity mentions. They can significantly improve search-result quality compared to just searching in documents directly about the entity: fewer relevant entities are missed and finding more hits for obvious cases helps ranking them better. For example, one can show results with the most prominent (most frequently mentioned) fires first. On top of that, results can be accompanied with better text snippets to display a compelling result with text passages that make it clear why a certain entity was returned.

Both example text snippets contain the crucial information about a fire, but they lack the information that the Pantheon is a *building* and that it is *located in Europe*. This is exactly where a knowledge base can be of great help. Consider the following excerpt from Freebase² (Bollacker et al., 2008):

Knowledge base excerpt

<Pantheon>	<Architect>	<Apollodorus_of_Damascus>	.
<Pantheon>	<is-a>	<Building>	.
<Pantheon>	<located-in>	<Europe>	.
<Panther>	<is-a>	<Animal>	.

It contains what is missing in the above text snippets to correctly identify the Pantheon as one of the results to the query. Such knowledge bases have become widely available in recent years. They are created manually as community efforts, automatically extracted from text, or as a combination of the two. The knowledge is usually represented as a collection of triples, each consisting of a subject, a predicate, and an object.

A suitable search engine has to combine the information from these two sources in an effective and efficient way. In Section 4.1 we discuss our system Broccoli (Bast et al., 2012b; Bast and Buchhold, 2013; Bast et al., 2014b) and how we establish this combination. A screenshot of the example query in Broccoli is depicted in Figure 1.

The Broccoli search engine enables its users to search for queries like the one from our example and thus effectively helps researching a topic. Its user interface and context-sensitive suggestions play an important part for that as we describe in more detail in Section 4.1.3. However, there are also downsides to this design: The query language

²The example actually uses the style of FreebaseEasy, our own sanitized version of Freebase with human-readable entity names. See Section 4.1.3 for how we derived it.

type here to extend your query ...

Words

Classes:

Location	(1143)
Structure	(367)
Tourist attraction	(104)

1 - 3 of 38

Instances:

Royal Opera House	(58)
Pantheon	(34)
Christiansborg Palace	(23)

1 - 3 of 276

Relations:

occurs-with	<Anything>
Architect	<Person> (189)
near-travel-destination	<Location> (94)

1 - 3 of 7

Your Query:

Building


occurs with
destroy* fire

located in
Europe

Hits:
1 - 2 of 276

Royal Opera House

Knowledge Base: Royal Opera House
Royal Opera House: is a **building**; located in **Europe**.
Document: Royal Opera House
On 5 March 1856, the **theatre** was again **destroyed** by fire.



Pantheon

Knowledge Base: Pantheon
Pantheon: is a **building**; located in **Europe**.
Document: Hadrian
In Rome, the **Pantheon**, originally built by Agrippa but **destroyed by fire** in 80, was rebuilt under Hadrian in the domed form it retains to this day.




Figure 1: A screenshot of our example query. The boxes on the left-hand side can be used to restrict or relax the query. Suggested items are context sensitive to the current query. The actual results, including snippets that explain why an item matches, can be found on the right.

contains features and restrictions that are mostly motivated by the user interface. Thus, while the search backend can be queried directly over HTTP and hence used as an API, the non-standard query language is a limiting factor, especially for its potential as a component in other systems, e.g., for question answering.

In contrast, limitations due to a non-standard query language are not an issue for classic knowledge-base queries that do not involve text search. There, SPARQL³ has evolved as a standard. It has a syntax similar to that of SQL and allows the specification of patterns which should be matched in the knowledge base. These patterns are expressed as a set of triples, where subject, predicate and object can be replaced by variables. For example, a query for buildings in Europe can be written as:

Query 1: Basic SPARQL

```

SELECT ?b WHERE {
  ?b <is-a> <Building> .
  ?b <located-in> <Europe>
}

```

³<http://www.w3.org/TR/rdf-sparql-query>

The result of this query is a list of buildings. If we had selected more than one variable, the result would be a list of tuples, where each matching combination would appear in that list as its own tuple.

Natively, SPARQL has no support for text search. Only regular expressions can be used to match entire literals in the KB. Several existing SPARQL engines have developed their own extensions to support keyword text search in those literals. However, they usually stop at this rather shallow combination of knowledge-base data and text and do not support a deep integration through entity occurrences. Consequentially, there is no support to search for co-occurrence between two entities. In particular, queries may not include variables (and thus subqueries) to match within text. In principle, it is possible to emulate a deeper combination by adding artificial predicates to the KB. We explain this in detail in Section 2.2. However, queries then quickly become very inefficient. We quantify this difference in performance when we evaluate our system in Section 4.2.4.

Therefore, we propose an extension to SPARQL. We simply add two special-purpose predicates: *ql:contains-word*, which allows words and prefixes to be linked to text records, and *ql:contains-entity*, which allows this linking for entities and variables. Thus, the query triples *?t ql:contains-word fire* and *?t ql:contains-entity ?x* mean that the word *fire* and the entity *?x* occur in text record *?t*. The first predicate is very similar to extensions that have been made to existing SPARQL engines, the second predicate, however, makes our query language significantly more powerful. Note that we only extend the query language but do not recommend explicitly adding these predicates to the knowledge base. For our system, we take a text corpus (in addition to the KB) with recognized entity occurrences as input and use the special-purpose data structures described in Section 4.2.3 to index the text for efficient retrieval.

We call this query language SPARQL+Text and formulate the example query as:

Query 2: SPARQL+Text

```
SELECT ?b TEXT(?t) WHERE {
  ?b <is-a> <Building> .
  ?b <located-in> <Europe> .
  ?t ql:contains-entity ?b .
  ?t ql:contains-word "destroy* fire"
}
ORDER BY DESC(SCORE(?t))
```

Apart from the special *ql:contains-* predicates, Query 2 displays further additions to the SPARQL language, that we have made for convenience. Without them, the variable *?t* from the example matches a numeric ID for fitting text records. While this is enough to answer the queries just fine and to retrieve the Pantheon and other relevant buildings, the additions can make the result a lot more useful. *TEXT(?t)* allows selecting matching

text passages for the text record variable $?t$ as result snippets. $SCORE(?t)$ yields a score for the text match that can be selected or used to obtain a proper result ranking, as done in the example query.

Like this, Query 2 can retrieve everything about *buildings in Europe that were destroyed in a fire* that is displayed as hits in Figure 1. The first two triple patterns in the query are responsible for using the knowledge base to restrict the answer to buildings in Europe, the last two patterns describe the text match to restrict the answer to entities that occur with the prefix *destroy** and the word *fire*.

We have developed QLever (pronounced “clever”), a query engine with full support for efficient SPARQL+Text search. Its novel index and query processing allows efficient answering of complex queries over billions of triples and text records. In Section 4.2 we describe this system and its technical contributions that allow highly efficient queries over very large combined data.

In the following we list the most important contributions of our work. For each of them we point the reader towards the section of this document that discusses the work in more detail. Our individual publications are listed in Section 3. The underlying research topics are presented in more detail in Section 4.

1.1 Contributions

With QLever and Broccoli, we have developed two fully-usable systems. The main focus of our work is on indexing and efficient query processing, but we have also tackled problems that improve the effectiveness and usability of the search.

QLever SPARQL+Text engine: We have developed a query engine with support for the SPARQL language with small but effective extensions which we call SPARQL+Text. Query times are faster or similar to those of state-of-the-art engines for pure SPARQL queries and faster by several orders of magnitude for SPARQL+Text queries. QLever is open source: <https://github.com/Buchhold/QLever> and still actively developed to this day. We cover our work on QLever in Section 4.2 of this document.

Broccoli search engine: We have developed the Broccoli search engine. Technically a predecessor to QLever, it only supports a subset of SPARQL (tree-shaped queries without variables for predicates). However, many additional features improve usability. For instance, Broccoli allows exploring knowledge bases due to its context sensitive suggestions for incremental query construction. It is available at <http://broccoli.cs.uni-freiburg.de>. The Broccoli search engine also features its own natural language processing and novel interface which are not part of this thesis but other lines of research conducted in our group. We cover our work on Broccoli in Section 4.1 of this document.

Indexing and Query Processing: The index data structures and algorithms behind Broccoli and especially QLever are by far the most efficient for SPARQL+Text queries. They are valuable on their own and may also find application in other systems than our

own, either directly or as adaptations of the main ideas behind them. In Section 4.2, we summarize these ideas and present the results of our evaluation.

Triple Scores: We have established a novel task and benchmark for computing relevance scores for KB triples with type-like predicates. Such a score measures the degree to which an entity “belongs” to a type. For example, Quentin Tarantino has various professions, including *Film Director* and *Actor*. The score for *Director* should be higher than the one for *Actor*, because that is what he is famous for, whereas as an actor, he mostly had cameo appearances in his own movies. These scores are crucial for ranking some queries within the Broccoli search engine (e.g., for a query for actors or a query for all professions of a person). Apart from an effective method for computing them, our research has led to the very lively triple scoring task (21 participating teams) at the 2017 WSDM Cup, see <http://www.wsdm-cup-2017.org/triple-scoring.html>. We cover our work on these scores in Section 4.3 of this document.

Freebase Easy: We have created a knowledge base that is derived from Freebase (Bollacker et al., 2008). The most obvious difference is our use of readable entity identifiers that make it possible for humans to directly look at and understand triples. This is impossible in the original. Our derivation also allows for simpler queries. Thus, it is used as KB in the current version of the public demo of the Broccoli search engine. The examples throughout this document, e.g., the KB excerpt in the introduction, use data from FreebaseEasy because of its great readability. We describe our work on FreebaseEasy and its application to Broccoli in Section 4.1.3 of this document.

Survey: We have published an extensive survey on *Semantic Search on Text and Knowledge Bases*. An important contribution is a classification of the numerous systems from this broad field according to two dimensions: the type of data (text, knowledge bases, combinations of the two) and the kind of search (keyword, structured, natural language). Following that classification, we identify and describe basic techniques that recur across the systems of a class as well as important datasets and benchmarks. We summarize this in Section 4.4 of this document.

2 Related Work

Semantic search is a broad field. We cover that in great detail in our extensive survey (Bast, Buchhold, and Haussmann, 2016). This section, more narrowly, relates our work to other approaches that efficiently answer queries over combined data. Such data consists of both: a knowledge base and text. The technical intricacies of particular systems are discussed later in this document, within their respective subsections of Section 4.

Here we distinguish four categories of related work: (1) approaches that answer (semi-) structured queries that contain parts to match in the KB and keywords to match in a text corpus, (2) SPARQL Engines that search knowledge bases and their extensions to text search, (3) systems for semantic web data and their particularities, (4) commercial web search engines and how they integrate knowledge base data to improve their results.

2.1 Search with (Semi-) Structured Queries

Broccoli is one of several systems that search in combinations of a text corpus and a knowledge base, where occurrences of entities from the KB have been linked (identified and disambiguated) in the text. Usually, the query language of such systems allows users to specify which parts of the query should be matched in the text and which parts should be matched in the KB. As such, a lot of the intelligence still has to be supplied by the user. A component that perfectly interprets and translates keyword or natural language queries would be the perfect addition and enable very powerful systems. Sadly, such a component is still up in the air.

Early systems use separate query engines for KB and text. A classic inverted index (usually from existing search engine software like Lucene⁴) is used for the text with the following addition: Just like for each normal word, there is also an inverted list of sorted document IDs for every entity. The KB part is handled by off-the-shelf SPARQL engines. Succeeding systems refine this idea but do not fundamentally deviate from it. For example, the strategy can be improved by adding additional inverted lists that represent entire classes of entities. In that case, there may be inverted lists for *buildings* or even *buildings in Europe*. There are many variants and extensions to this general idea. We describe concrete systems and their strengths and drawbacks w.r.t. efficiency in Section 4.2.2 and experimentally compare these ideas to our work in Section 4.2.4.

An important difference between all prior systems and our work is that the methods based on a classic inverted index only yield document centric results. Thus, for our example query, such systems would return a list of matching documents or snippets which state that a building in Europe was destroyed in a fire. It is left to the user to figure out which buildings are mentioned. It is also entirely possible that many of those hits talk about the same building, making it very hard for a user to obtain a proper list.

⁴<https://lucene.apache.org/core/>

Broccoli and QLever can return both, lists of matching documents and lists of matching entities. Further, we are also able to provide sensible rankings for either. Note how properly ranking a list of possible result entities is entirely impossible when query results are documents rather than entities. Our work on indexing and query processing (described in Section 4.2) makes this possible with roughly the same efficiency as classic keyword search with a classic inverted index.

Compared to different lines of research (e.g., those described in the following subsections), systems following this approach have huge potential but several drawbacks. Result quality is very high (see Section 4.1.4). However, to achieve that, it is necessary that a user is able to ask the perfect query (or some component is able to infer it). Further, text and knowledge base have to be linked perfectly and contain the necessary information.

Another issue are non-standard query languages. So far, there is no standard for semi-structured queries with parts supposed to match in the KB and parts supposed to match in a text corpus. Our system QLever comes close by supporting the SPARQL language with a very small, but powerful, set of extensions.

2.2 SPARQL Engines

Popular systems for SPARQL queries, like Sesame (Broekstra, Kampman, and Harmelen, 2002) or Jena⁵, are built as layers on top of systems that actually store the triples, often relational databases. As all SPARQL queries can be rewritten to SQL; see the work by Elliott et al., (2009), support for them has been added by most relational databases. In contrast to classic relational databases, so-called triple stores are purpose-built for storing the triples that comprise knowledge-base data. Consequently, efficiency of such SPARQL engines vastly depends on these underlying systems. In Section 4.2.2 we describe the most efficient systems from industry and research and their relation to our work on efficient indexing and query processing.

What distinguishes our work (apart from very fast query times) is the deep integration of text search. The SPARQL language does not include keyword search natively. It does, however, allow literals from the knowledge base to be filtered by regular expressions. However, complex regular expressions to filter on textual literals are prohibitively inefficient to use for keyword search on large amounts of text.

Since classic keyword search is often a highly useful feature, it has been added by several SPARQL engines and frameworks. Usually, they introduce a special predicate that allows keyword search in KB literals. In Jena, this predicate is called *text:query* and in the efficient triple store Virtuoso⁶, which we describe in more detail in Section 4.2.2, it is called *bif:contains* (where *bif* means *build-in function*). With this, queries like the following can be answered:

⁵<https://jena.apache.org>

⁶<https://virtuoso.openlinksw.com/>

Query 3: Example bif:contains

```
SELECT ?b WHERE {  
  ?b <is-a> <Building> .  
  ?b <located-in> <Europe> .  
  ?b <description> ?d .  
  ?d bif:contains "'destroy*' 'fire'"  
}
```

Notice how this is not identical to our example from above. It relies on the knowledge base having description literals for building entities and that those contain the necessary information. It is not intended to search for occurrences of entities anywhere in a large text corpus. For our example, the fact that the Pantheon was once rebuilt after being destroyed in a fire, is not mentioned in the description of the Pantheon in the Freebase KB (Bollacker et al., 2008).

However, it is possible to fully emulate SPARQL+Text Search. Three possibilities are depicted in Figure 2.

Emulating SPARQL+Text Search with SPARQL Engines

I. Without extensions:

<record:123>	<contains-entity>	<Pantheon>	.
<record:123>	<contains-entity>	<Augustan_Age>	.
<record:123>	<contains-word>	<word:destroyed>	.
<record:123>	<contains-word>	<word:fire>	.
...

II. With keyword search in literals (loses expressiveness):

<Pantheon>	<text>	"The Augustan Pantheon was destroyed along with other buildings in a huge fire in 80"	.
<Augustan_Age>	<text>	"The Augustan Pantheon was destroyed along with other buildings in a huge fire in 80"	.

III. With keyword search in literals (lossless):

<record:123>	<contains>	<Pantheon>	.
<record:123>	<contains>	<Augustan_Age>	.
<record:123>	<text>	"The Augustan Pantheon was destroyed along with other buildings in a huge fire in 80"	.

Figure 2: Three emulation strategies that enable classic SPARQL engines to answer SPARQL+Text queries (with limited efficiency).

The first possibility just uses standard SPARQL without any extensions. We simply add all text records and words as entities to the KB and add a triple with an explicit *contains-entity* and *contains-word* predicate for every word- and entity-occurrence. The second possibility makes use of the common extensions for keyword search in literals which we have mentioned above. We simply connect each entity to a literal that represents the entire text literal. However, this strategy has a drawback: the information that the entities <Pantheon> and <Augustan_Age> co-occur is lost. In fact, all information about co-occurrence between entities is lost in this way. The third strategy overcomes this issue. We now introduce an explicit entity for each text record and connect it to its text literal with another triple.

Neither strategy is perfectly suited for efficiency. We compare the performance of QLever against state-of-the-art systems on pure SPARQL queries and SPARQL+Text emulated this way in our publication of QLever (Bast and Buchhold, 2017) and present an excerpt with the main results in Section 4.2.4. While QLever is also faster on many classic SPARQL queries, the gap widens significantly when its native support for SPARQL+Text is compared against the emulations.

In terms of result quality, the above strategies emulate (parts or all of) the search capabilities of our systems QLever and Broccoli. They are not as powerful w.r.t. user convenience, because retrieving result snippets and the number of matches would still have to be added separately, but overall, the emulations deliver the same results as our systems. This quality is examined in Section 4.1.4 and especially in our publication (Bast, Buchhold, and Haussmann, 2017). A different take on SPARQL+Text has been demoed under the name RDF Xpress (Elbassuoni, Ramanath, and Weikum, 2012) that supports approximate matching of string literals via a special-purpose language model (Elbassuoni et al., 2009). Efficiency does not play a major role for that work and the overall search paradigm is similar to the text extensions of Virtuoso and Jena. However, it would still be interesting to compare the advanced ranking function and its implications for result quality on our typical queries. Sadly, we could not find publicly available software to do so.

Of course, our work does not fully subsume previous work on SPARQL engines. These engines deal with important topics that we ignored for QLever so far: Data inserts and thus incremental index updates, as well as distribution across several machines are very important topics. While we do not see principal problems that would make this impossible, the process is not trivial either and we have not concerned ourselves with it yet. There is also a large body of research on reasoning on knowledge-base data (e.g., infer subclasses information on the fly) and on certain data analysis tasks (e.g., how two entities connected over n hops in the data graph). Specialized systems can naturally outperform our work on those queries.

2.3 Semantic Web Search

The Semantic Web contains a vast amount of structured and semi-structured data. The data from the Semantic Web is often also called *linked open data (LOD)*, because contents can be contributed and interlinked by anyone. This happens in two ways. First, as documents with triples (similar to the example knowledge base excerpts used throughout this document) that can link to each other, just like ordinary web pages can link to each other. Secondly, through semantic markup embedded in web pages. For example, a website about a movie that mentions its director and actors can make the information readable to machines by adding additional tags to the HTML source that are not shown to a reader. There are multiple widely used formats for semantic markup. We describe them in Section 2.1 of our survey (Bast, Buchhold, and Haussmann, 2016).

As such, the Semantic Web also consist of both, textual and structured knowledge-base data. Searching it thus initially seems to be very closely related to our work. However, a closer look reveals that systems for semantic web search usually have to focus on very different problems than we do.

The nature of the Semantic Web leads to two major challenges for search: (1) The sheer amount of data⁷ requires very efficient solutions and (2) the absence of a global schema (information about the same entity can use different names or identifiers of that entity and different predicates can have the same or similar meanings) makes structured languages like SPARQL only suited for querying subsets (in which the schema is somewhat consistent), if at all.

Therefore, typical systems for semantic web search usually process that data in ways that are similar to classic keyword- and web search. The basic idea is to store all data from triples in an inverted index. A (virtual) document per subject entity is created, that consists of all corresponding predicates and objects as text. Then, keyword queries are issued and results are ranked like for classic keyword search. This is very efficient, and works despite the inconsistent schema problem. Well-researched ranking techniques for keyword search can help to improve search quality. However, it still does not compare to the precise query semantics offered by SPARQL queries on consistent knowledge bases. Semplore (Wang et al., 2009) is a typical system following this approach.

More advanced systems aim at improving search quality while retaining high performance. Blanco, Mika, and Vigna, (2011) examine several variants to index the virtual documents. In particular, valuable information is lost if predicates are dropped entirely or just mixed with objects in the virtual documents. They use the fielded inverted index of MG4J (Boldi and Vigna, 2005) and examine trade-offs between query time and result quality. Such a fielded index and BM25F (Zaragoza et al., 2004), which is an extension of the well-known BM25 ranking function, were originally developed for classic keyword search and to let matches in titles or URLs contribute more to the score of a document

⁷While there are no current estimates available, the subsets examined by Meusel, Petrovski, and Bizer, (2014) and Guha, Brickley, and MacBeth, (2015) suggest that the number of triples in form of semantic markup has reached hundreds of billions.

than matches somewhere in the body. This can be applied to semantic web search in several ways. With dedicated fields for subject, predicate, and object, the structure is retained – at the cost of query time. However, the preferred way for their use-case discards most of this structure for the sake of efficiency: Index fields are only used to group triples by predicate importance and to boost the important ones in the ranking function. For standard keywords queries, this allows queries that are as fast as for a vanilla inverted index but with significantly better result quality.

Siren (Delbru, Campinas, and Tummarello, 2012) is another system in that vein. It is built on top of the popular text search engine library Lucene⁸ and supports queries that correspond to star-shaped SPARQL queries. There is one variable at the center and several triples can be connected to it, but not longer chains of triples. In those queries, predicate and object names can be matched via keyword queries. There are inverted lists for words in predicate names and for words in object names. Each index item contains information about the triple to which it belongs, namely: the ID of the subject entity, the ID of the predicate, the ID of the object (only for words in object names), and the position of the word in the predicate or object. Standard inverted list operations can then be used to answer a query for all entities from triples containing, e.g., *author* in the predicate name, and *john* and *doe* in the object name.

In summary, systems for searching the Semantic Web may solve a similar problem to ours as they answer queries over combined data that consists of triples and text alike. However, the different circumstances make the typical techniques very different. Searching the large and inconsistent Semantic Web is much harder and thus systems aim for much less precise results.

2.4 Commercial Search Engines and KBs

As we have argued in the introduction, the queries we answer are very relevant to actual users and they can be answered very well with knowledge bases and text. Thus, commercial web search engines obviously also follow this approach. Unfortunately, there are no publication that disclose their entire efforts. But from the user experience and existing publications it is possible to make an educated guess.

We focus on Google Search⁹ as of 2017. We believe that a query is classified and answered by several subsystems. Depending on the individual results and the confidence in them, the final result page is compiled. While it always contains the well-known 10 blue links as the result of keyword search, often there is an extra section at the top with relevant information from one of the special subsystems. There are hand-compiled answers showing nicely edited information for many queries. For example, for the query *actors won oscar*, or during events like the last FIFA World Cup, such compilations are displayed prominently. In the following, we focus on automated efforts to answer semantic queries.

⁸<http://lucene.apache.org>

⁹<https://www.google.com>

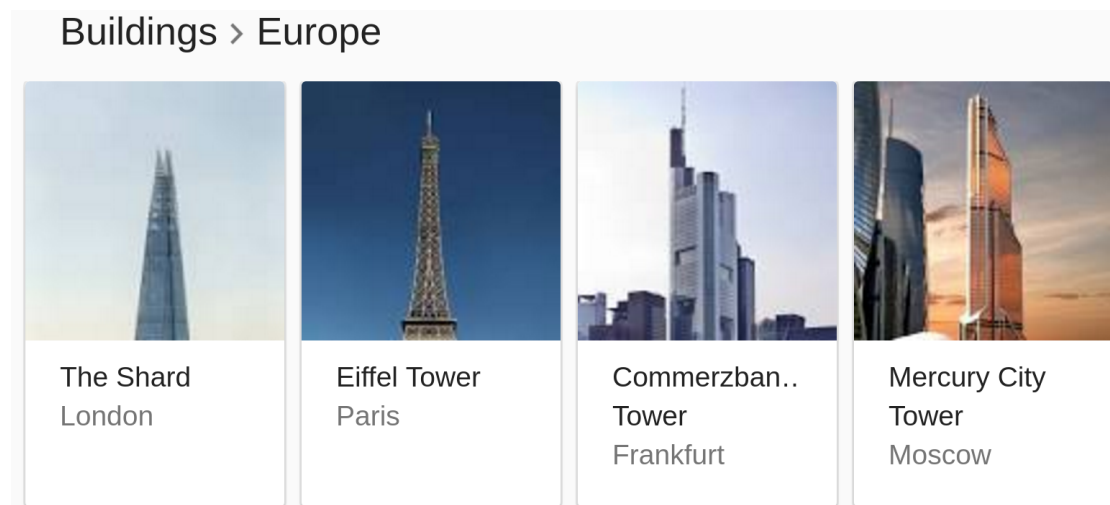


Figure 3: **Additional section at the top of the search result page for the query *buildings in Europe* from Google Search Jun 2017**

The query *buildings in Europe* displays pictures of popular buildings together with the city in which they are located. Figure 3 shows this augmentation of the result page. As the top indicates, it is a list of buildings restricted on Europe. This suggests that the query is answered with the help of a knowledge base.

Also, Google has been rather transparent on their efforts concerning knowledge bases and published several papers. In 2010 Google acquired Metaweb, the company behind the Freebase KB. While Freebase (Bollacker et al., 2008) consists of manual user-submitted contributions and semi automated integrations of other publicly available data (e.g., from GeoNames¹⁰ or MusicBrainz¹¹), Google has continued development of its internal knowledge base, the Knowledge Graph (Singhal, 2012). More recently, they have also published research on automatic knowledge-base construction. Their Knowledge Vault (Dong et al., 2014) is an effort to extract triples from web content and to fuse the results from several extractors and previously known triples. Their KB data delivers precise results for many queries and its value is apparent for many queries.

We have seen this in Figure 3 for the query *buildings in Europe* and many queries work similarly, e.g., *american actors* or *director martian*. Even formulations as natural language questions are answered in the same way, e.g. *Who directed the Martian?*

Additionally, some queries directly find tables that answer the query very well. For example, the query *french actors who won an oscar* directly matches a table with that exact content. We attribute this to the work behind WebTables (Cafarella et al., 2008).

¹⁰<http://www.geonames.org>

¹¹<http://linkedbrainz.org/rdf/dumps/20150326/>

Google also answers natural language questions from annotated text. This is usually done in an extractive way: the answer is found in text with annotated entities. This happens, for example, for queries like *When was the first antibiotic discovered?* for which Google currently outputs:

1928

But it was not until **1928** that penicillin, the first true antibiotic, was discovered by Alexander Fleming, Professor of Bacteriology at St. Mary's Hospital in London.

Especially, this extractive question answering from text and the queries from before, that can be answered directly from the KB, already accomplish similar things as our kind of search. However, we cannot observe anything for complex queries and see no answers that are derived from combinations of KB and text, yet. For the query from our introduction, *buildings in Europe that were destroyed in a fire* we just get textual matches and, in general, results that are not very satisfying. For example, the search returns documents about buildings that were destroyed in WW2 or London buildings that were destroyed in a fire.

In summary, Google Search has many powerful subsystems at its disposal and already works very well for many queries. However, they obviously have to be relatively conservative with new features and only introduce them once they are very solid, in order to not jeopardize the search experience. Thus our work goes significantly further than current commercial search engines when searching on combined data. Of course, an integration of another subsystem to search combined data is not easy – but it is absolutely not unthinkable to happen in one way or the other within the near future.

3 Publications

In the following two subsections we present publications that have already been published in, or accepted to, peer reviewed conferences and journals, and one publication that has not gone through peer review but still describes relevant parts of our work. Within each section, publications are listed in chronological order. For each publication, we provide a short description of its kind (full paper, demo paper, extensive survey) and contents. We also specify what part of the work can be attributed to which of the authors (as authors are usually in alphabetical order by convention).

3.1 Peer-Reviewed Publications

In the following, we list our peer-reviewed publications. Two papers have been accepted for inclusion in the conference proceedings and a journal respectively, but have not yet been published.

A Case for Semantic Full-Text Search

SIGIR-JIWES 2012 (Bast et al., 2012a)

Hannah Bast, Florian Bärle, Björn Buchhold and Elmar Haussmann

Position paper that introduces the Semantic Full-Text Search paradigm for search in text and knowledge bases. We argue how knowledge bases and text corpora both have their strengths and weaknesses but can complement one another very well.

The paper is covered in Section 4.1 of this document.

All authors wrote the paper and conducted the research that backs up the ideas presented in this position paper.

An Index for Efficient Semantic Full-Text Search

CIKM 2013 (Bast and Buchhold, 2013)

Hannah Bast and Björn Buchhold

Full research paper that presents and evaluates the index behind the Broccoli search engine. The index is tailor-made from scratch to efficiently support search in text linked to a knowledge base. This yields significantly faster query times than previous work.

The paper is covered in Sections 4.1 and 4.2 of this document.

Both authors conducted the research. Björn Buchhold provided all implementations. Both authors designed the evaluation benchmark, Björn Buchhold conducted the evaluation. Both authors wrote the paper.

Easy Access to the Freebase Dataset

WWW 2014 (Bast et al., 2014a)

Hannah Bast, Florian Bärle, Björn Buchhold and Elmar Haussmann

Demo paper that presents our publicly-available adaption of the Freebase KB. We derive a version that can be searched more easily and whose triples are readable for humans. We provide a web application that offers convenient access.

The paper is covered in Section 4.1 of this document.

Hannah Bast, Björn Buchhold and Elmar Haussmann conducted the research. Björn Buchhold and Elmar Haussmann implemented the ideas. Florian Bärle adapted the user interface of Broccoli for the web application. Hannah Bast, Björn Buchhold and Elmar Haussmann wrote the paper.

Semantic Full-Text Search with Broccoli

SIGIR 2014 (Bast et al., 2014b)

Hannah Bast, Florian Bärle, Björn Buchhold and Elmar Haussmann

Demo paper that presents the Broccoli search engine, its interactive user interface, and public API.

The paper is covered in Section 4.1 of this document.

All authors conducted the research on the search paradigm and general system design. All authors wrote the paper.

Relevance Scores for Triples from Type-Like Relations

SIGIR 2015 (Bast, Buchhold, and Haussmann, 2015)

Hannah Bast, Björn Buchhold and Elmar Haussmann

Full research paper that describes how to compute relevance scores for knowledge base triples. The scores can be used to properly rank results of entity queries on a knowledge base. We present and evaluate several models to learn these scores from a large text corpus and design a crowdsourcing task to create a benchmark for triple scoring.

The paper is covered in Section 4.3 of this document.

All authors conducted the research, designed the crowdsourcing experiment and the evaluation. Björn Buchhold and Elmar Haussmann provided all implementations. All authors wrote the paper.

Semantic Search on Text and Knowledge Bases

FnTIR 2016 (Bast, Buchhold, and Haussmann, 2016)

Hannah Bast, Björn Buchhold and Elmar Haussmann

Extensive survey (156 pages) over the huge field of semantic search on text and knowledge bases.

The paper is covered in Section 4.4 of this document.

All authors contributed in deciding the overall structure and scope of the survey. All authors surveyed the literature and prepared summaries for systems to include or exclude. All authors wrote the survey.

WSDM Cup 2017: Vandalism Detection and Triple Scoring
WSDM 2017 (Heindorf et al., 2017)

Stefan Heindorf, Martin Potthast, Hannah Bast, Björn Buchhold and Elmar Haussmann

Overview paper for the WSDM Cup 2017 and its two tasks, vandalism detection and triple scoring.

The paper is covered in Section 4.3 of this document.

Stefan Heindorf and Martin Potthast organized the vandalism detection task. Hannah Bast organized the Triple Scoring task. Hannah Bast, Björn Buchhold and Elmar Haussmann established the triple scoring task, created a benchmark via crowdsourcing and defined sensible evaluation metrics. Stefan Heindorf, Martin Potthast and Hannah Bast wrote the paper.

QLever: A Query Engine for Efficient SPARQL+Text Search
Accepted to CIKM 2017 (Bast and Buchhold, 2017)

Hannah Bast and Björn Buchhold

Full research paper that describes the QLever query engine for efficient SPARQL+Text search. It introduces a novel knowledge-base index, improves upon our text index from (Bast and Buchhold, 2013) and introduces new algorithms for planning and executing SPARQL+Text queries.

The paper is covered in Section 4.2 of this document.

Both authors conducted the research. Björn Buchhold provided all implementations and conducted the experiments. Both authors wrote the paper.

A Quality Evaluation of Combined Search on a Knowledge Base and Text

Accepted to KI Journal (Bast, Buchhold, and Haussmann, 2017)

Hannah Bast, Björn Buchhold and Elmar Haussmann

Research paper that describes a detailed quality evaluation and error analysis of our search paradigm.

The paper is covered in Section 4.1 of this document.

All authors designed the evaluation and analysed results. Björn Buchhold and Elmar Haussmann performed most of the manual evaluation and error analysis. All authors wrote the paper.

3.2 Other Publications

In the following, we list a publication that has not been published in peer-reviewed proceedings. However, we still consider it relevant to this thesis and find it important to adequately attribute author contributions to the Broccoli system. Further, it provides a good overview of the research that underlies some of the peer-reviewed publications from before.

Broccoli: Semantic Full-Text Search at your Fingertips

CoRR 2012 (Bast et al., 2012b)

Hannah Bast, Florian Baurle, Björn Buchhold and Elmar Haussmann

Research paper that describes the Broccoli system and all of its major components. These components includes the basic idea behind its efficient index, its natural language processing (contextual sentence decomposition) and its interactive user interface.

The paper is covered in Section 4.1 of this document.

All authors conducted the research on the search paradigm and general system design. Hannah Bast and Björn Buchhold conducted the research on efficient indexing and query processing, Björn Buchhold implemented it. Hannah Bast and Elmar Haussmann conducted research on the contextual sentence decomposition, Elmar Haussmann implemented it. All authors designed the user interface, Florian Baurle implemented it. Florian Baurle, Björn Buchhold and Elmar Haussmann implemented a data preprocessing pipeline to load a text corpus and a knowledge base into the search system. All authors performed the evaluation and wrote the paper.

4 Research Topics

In the following we summarize the research topics to which we contributed while developing our systems.

4.1 Semantic Full-Text Search with Broccoli

The Broccoli search engine allows what we call *Semantic Full-Text Search*. The query language is closely tailored towards the user interface (recall Figure 1 from the introduction). Strictly speaking, it is a subset of SPARQL (restricted to tree-shaped queries with exactly one selected variable at the root, and not using variables for predicates) but extended by the special *occurs-with* predicate. This predicate can be used to specify co-occurrence of a class (e.g., *building*) or instance (e.g., *Pantheon*) with an arbitrary combination of words (e.g. “destroyed fire”), other instances (e.g., *Rome*), and/or further sub-queries (e.g., *roman emperors that were assassinated*).

Technically, this covers only a subset of the SPARQL+Text search described in the introduction (which, in contrast, is fully supported by our more recent system, QLever, which we cover in Section 4.2). However, we have gone great lengths to provide an appealing user experience and to improve the quality of the search results of Broccoli.

In our publication (Bast et al., 2012a), we have first argued that we want to leverage the strengths of both, knowledge bases and text, to overcome the weaknesses of each other and our system, Broccoli, applies this idea in practice. In addition, its semantic full-text search paradigm also emphasizes the importance of suggestions for construction and meaningful result snippets. Context-sensitive suggestion during query construction are absolutely crucial as users cannot be expected to know the exact names of the entities and their relations in the knowledge base. Meaningful result snippets are equally important. In the following, we consider an example query in order to demonstrate how essential these snippets can be.

The system owes its name *Broccoli* to a query for *plants with edible leaves* where it matches entities that, according to the KB, are plants and that occur together with the words *edible* and *leaves* in the text corpus. This correctly finds broccoli as one of the results. However, such a search may also yield incorrect answers: Imagine a text passage like “*Rhubarb stalks are edible, but its leaves are toxic.*” which also has occurrences of a plant and the two words we are looking for.

In Section 4.1.3 we will briefly explain how natural language processing can help to avoid these mistakes in the first place. Still, we cannot guarantee perfect precision. If our system returns a long list of matching plants, somewhere down the list, false positives are bound to appear. However, with the help of good result snippets, a user can quickly assess the reliability of each hit with minimal effort (and thus will hopefully not eat rhubarb leaves because of our search engine).

4.1.1 Problem Statement

We want to make queries like the example *buildings in Europe that were destroyed in a fire* possible and allow users to conveniently retrieve this information from combinations of knowledge base and text. We are dedicated to build a system, where a user does not require prior knowledge of either KB or text corpus and query results should be of the highest quality possible. In addition, we want transparency: If it is obvious to the user why any particular hit has been returned, false positives are much less harmful than if they were provided by a black-box system.

4.1.2 Related Work

We have discussed other systems for search with (semi-) structured queries on combinations of text and knowledge base in Section 2.1 and will elaborate on their indexing and query processing further in Section 4.2.2.

Here, we want to differentiate our work on Broccoli and Semantic Full-Text Search from lines of work that answer similar queries, but approach the problem in a different way. Most notably, there have been two benchmarks that feature tasks whose queries resemble ours: the TREC Entity Track and the SemSearch Challenge.

The TREC Entity Track (Balog et al., 2009; Balog, Serdyukov, and Vries, 2010) featured queries searching for lists of entities, just like in our work. They are particularly interested in lists of entities that are related to a given entity in a specific way. Thus, the task is called "Related Entity Finding". This means that they focus on a particular subset of the queries supported by Broccoli (and by extension also by QLever). A typical query is *airlines that currently use boeing 747 planes*. Along with the query, the central entity (*boeing 747*) as well as the type of the desired target entities (*airlines*) were given. The benchmark predates our work and was not continued, but we have used the queries in our evaluation described in Section 4.1.4 to demonstrate the strengths of Broccoli and its search paradigm.

In 2011, the SemSearch Challenge (Blanco et al., 2011) featured a task with keyword queries for a list of entities (for example, *astronauts who landed on the moon*). These queries are of the same kind as ours and we have used them to evaluate both, the result quality (see Section 4.1.4) and efficiency (see Section 4.2.4) of our work.

We provide more details on those benchmarks in our survey (Bast, Buchhold, and Haussmann, 2016). The major difference to our work is the kind of query. The systems that competed on these benchmarks all work with keyword queries, whereas our queries build upon explicit knowledge-base facts as known from SPARQL. This also leads to different techniques that are being used. In systems that ran on those benchmarks, entities are usually associated with a bag of words (e.g., through virtual documents as we have described for semantic web search in Section 2.3) and then ranked for the keyword query.

Type information (e.g., *airlines* in the example above) is usually only used to filter results and systems do not make use of advanced knowledge-base data.

In contrast, our systems require a structured query. While this can be considered a limitation, if we are provided with such a query, the search is more powerful and can yield results of higher quality as we point out in our evaluation in Section 4.1.4.

4.1.3 Approach

The Broccoli system and all its components are described in our publications (Bast et al., 2012b), (Bast and Buchhold, 2013) and (Bast et al., 2014b). Here, we provide a high-level overview and elaborate on unpublished improvements to our context-sensitive query suggestions which allow iterative query construction. We also describe FreebaseEasy, our own adaption of the Freebase (Bollacker et al., 2008) knowledge base, which has originally been published in (Bast et al., 2014a).

In principle, Broccoli can be set up for any knowledge base and text corpus. Our preprocessing is organized in a pipeline and its components can be adjusted for the input at hand. We use Apache UIMA¹² to manage our pipeline. Thus, each component has a clearly defined interface and we specify which data it produces or modifies. Further, UIMA allows us to easily scale our preprocessing to multiple processors and across several machines through its Asynchronous Scaleout capabilities.

In practice, we have put a large focus on our components for Wikipedia text and we show our preprocessing for this input in Figure 4. We parse an official Wikipedia dump (in XML format) to obtain the full text of each article and to transform knowledge about sections boundaries, linked Wikipedia pages (and thus entities), and similar information from markup into external data structures that reference positions in the text. After running a tokenizer, we use third-party software (configurable) to perform a constituent parse that also gives us part-of-speech tags. This information is needed later by the entity recognizer and especially by further NLP steps. Afterwards, we link all direct and indirect mentions of entities to their respective entries in the KB. In Wikipedia articles, first occurrences of entities are already linked to their Wikipedia page. Whenever a part, a synonym, or the full name of that entity is mentioned again in the same section (or one of its subsections), we recognize it as that entity. When we encounter references (anaphora) we assign them to the best matching entity. Pronouns, like *he*, *she*, *it*, *his* or *her*, are assigned nearest preceding entity of matching gender and mentions of *the* *<type>*, e.g., *the building*, to the last matching entity of that type. For text corpora other than Wikipedia, state-of-the art approaches for named entity recognition and disambiguation, such as Wikify! (Mihalcea and Csomai, 2007) or the work by Cucerzan, (2007) or Monahan et al., (2014) can be used instead.

Finally, we obtain semantic contexts from the text. We employ contextual sentence decomposition (Bast and Haussmann, 2013), an approach for open information extraction,

¹²<https://uima.apache.org/>

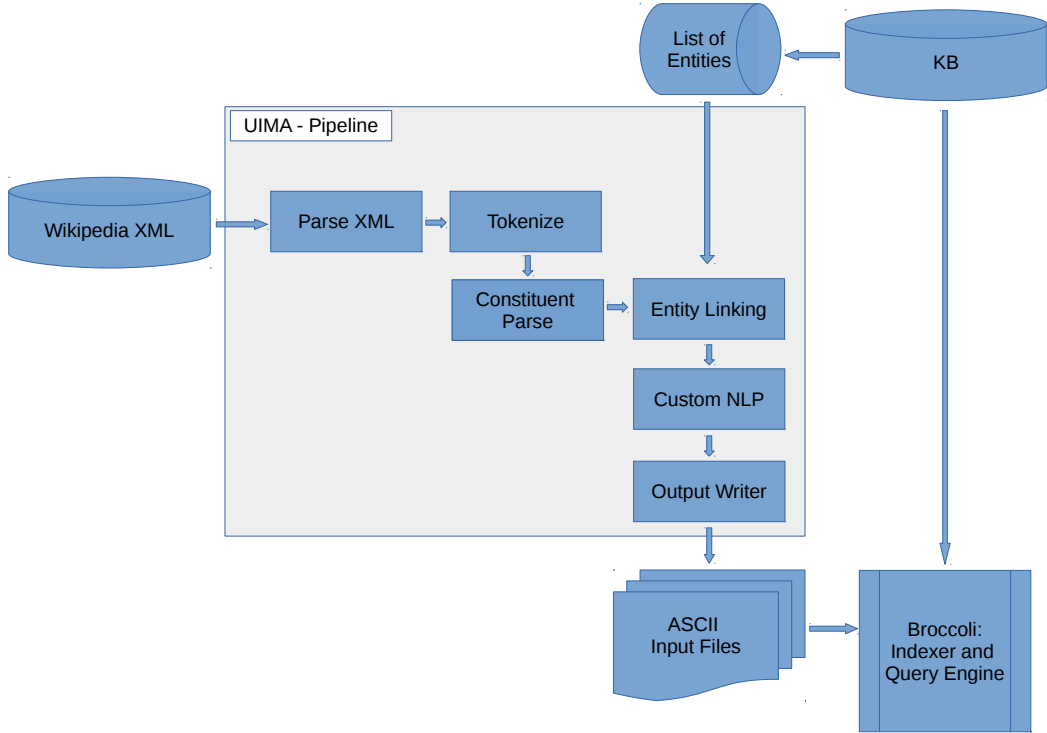


Figure 4: **The pre-processing pipeline to create an instance of the Broccoli search engine from a KB and a Wikipedia XML text corpus, slightly simplified.**

to split sentences into multiple contexts. We also unite items inside enumerations with the sentence preceding the enumeration. Intuitively, the words inside such a context semantically belong together. The special *occurs-with* predicate of the query language of Broccoli specifies co-occurrence within these semantic contexts. We have also experimented with other text segmentations, e.g., using sentences, paragraphs or entire documents and demonstrate the benefit of the semantic contexts in (Bast, Buchhold, and Haussmann, 2017).

We then produce input from which we can build our index and process queries as described in (Bast and Buchhold, 2013). This is a crucial part of the Broccoli system, but we defer this for now and describe our work on indexing in more detail in Section 4.2.

Iterative query construction with context-sensitive suggestions

One important part of Broccoli is how its user interface (UI) guides the user through its context-sensitive suggestions. This allows iterative construction of complex queries,

without prior knowledge of the KB and how exactly certain predicates, types or entities are called.

Building	occurs-with	tallest
architect, the RELATION		
▼ Instances:		
Burj Khalifa	(340)	
Empire State Building	(203)	
Chrysler Building	(104)	
1 - 3 of 267		
▼ Classes:		
Location	(267)	
Structure	(267)	
Skyscraper	(226)	
1 - 3 of 41		
▼ Relations:		
located-in	<Location>	(252)
architect	<Person>	(197)
floors	<Integer>	(88)
1 - 3 of 8		

Building	occurs-with	destroy* fire
architect, the RELATION		
▼ Instances:		
Royal Opera House	(58)	
Old Parliament Building (Quebec)	(52)	
Pantheon	(34)	
1 - 3 of 167		
▼ Classes:		
Location	(167)	
Structure	(167)	
Tourist attraction	(26)	
1 - 3 of 47		
▼ Relations:		
located-in	<Location>	(122)
architect	<Person>	(97)
opened	<Date>	(82)
1 - 3 of 8		

Figure 5: **Context sensitive suggestion for incremental query construction in the Broccoli search engine.**

Figure 5 compares the context sensitive suggestions for highly similar, yet different two queries. The query is depicted at the top and suggestions to incrementally extend or refine it are shown in the three boxes. We distinguish instances (the entities matching the current query), classes (super- and subclasses of matching instances) and relations (KB predicates that are available for as many of the matching instances as possible). In the concrete example, both queries are about buildings and thus all suggestions are specific to that. However, there is also a difference between the two, due to their text part (shown in yellow in the query above). Buildings that occur with the word *tallest* tend to be skyscrapers, and have relations like the number of their floors, whereas buildings that occur with *destroy* fire* are more often historic buildings. The important benefit is that a user can find relevant relations even if she is not sure how exactly they are called in the KB.

How we compute these suggestions is described in detail in (Bast and Buchhold, 2013). In a nutshell, we solve the current query and then compute an additional join with our *is-a* (type) relation to fill the classes box. To fill the relations box, we compute a join with an artificial *has-relations* relation, that lists, for each entity, all relations it engages in. After the joins we aggregate the types (or relations) and rank them by their counts, i.e. for how many result instances they are available.

In practice, Broccoli uses another trick to improve the efficiency of these join-and-aggregate operations. As this trick has not been published so far, we describe its technical

details here for the first time. The intuition behind the trick is that many instances share the same set of types and relations. E.g., all buildings are also classified as *structure* and *location*. While the most famous ones in our KB have additional types, e.g., *skyscraper*, *tourist attraction*, etc., the long tail of less popular buildings has exactly these three types. This becomes even more significant for types with many instances like the 11 million *musical recordings* where most not only have the exact same set of types, but also the same set of relations they engage in.

We make use of this phenomenon and collect the most frequent set of types and connected relations as patterns and assign IDs to them.

Efficient Relation Suggestions			
Original content of has-relations:			
Burj Khalifa (ID 2)	has-relations	architect (ID 1)	
Burj Khalifa (ID 2)	has-relations	floors (ID 7)	
Burj Khalifa (ID 2)	has-relations	structural-height (ID 10)	
Dubai (ID 4)	has-relations	country (ID 3)	
Dubai (ID 4)	has-relations	population (ID 9)	
Eiffel Tower (ID 5)	has-relations	architect (ID 1)	
Eiffel Tower (ID 5)	has-relations	structural-height (ID 10)	
Empire State Building (ID 6)	has-relations	architect (ID 1)	
Empire State Building (ID 6)	has-relations	floors (ID 7)	
Empire State Building (ID 6)	has-relations	structural-height (ID 10)	
Paris (ID 8)	has-relations	country (ID 3)	
Paris (ID 8)	has-relations	population (ID 9)	
(1) Pattern ID to relations:			
1	1, 7, 10	Pattern 1	architect + floors + structural-height
2	3, 9	Pattern 2	country + population
(2) Entity ID to pattern ID:			
2	1	Burj Khalifa	Pattern 1
4	2	Dubai	Pattern 2
6	1	Empire State Building	Pattern 1
8	2	Paris	Pattern 2
(3) Has-relations without entities covered by patterns:			
5	1	Eiffel Tower	architect
5	10	Eiffel Tower	structural-height

Figure 6: **The Trick used by Broccoli to further speed-up relation suggestions. Here we assume unrealistically few relations for the sake of a clear example with only two patterns.**

We then index three things as shown in Figure 6: (1) a mapping from pattern ID to represented relations or types, (2) the pairs of entity ID and pattern ID, and (3) the classic KB predicates between entity ID and relation/type ID, but only for those entities with infrequent (or unique) patterns. Note that only entities that exactly match a relation or type pattern make use of them. All others, even with very slight deviations, are kept in the classic lists (3). Like this, we can now join the list of instances matching the query with lists (2) and (3). These two lists are, in combination, still much shorter than the original list where each entity would have many associated types/relations. In the concrete example from Figure 6, we can think of a query for buildings that yielded the result list [2, 5, 6]. We now join this list with lists for (2) and (3) that have sizes 4 and 2. Even if the effect is not very apparent in the small example, we can still observe that this is better than joining with the original list of size 12.

To compute the final suggestions, we can then simply iterate over matched patterns and accumulate the counts for each of the relations (or types) represented by the patterns. The trick works so well, because we are only interested in counts for each relation (or type) and not the pairs of instance and relation/type. Those pairs can instead be obtained by a separate query after the suggestion has been added to the current query.

Adaption of the Freebase KB

The current demo of Broccoli (broccoli.cs.uni-freiburg.de) uses FreebaseEasy, a knowledge base derived from Freebase (Bollacker et al., 2008). The original Freebase KB offers the largest number of useful facts out of all publicly available general-knowledge KBs. Size is important. For example, in early versions of Broccoli we worked with the original YAGO KB (Suchanek, Kasneci, and Weikum, 2007). However, if a query for *movies with Johnny Depp* retrieves only six movies, such results are not what a user should expect. Unfortunately, the larger Freebase KB is, unlike YAGO, not well suited for our kind of search in its unmodified state for three reasons:

1. Entities are identified by machine identifiers such as *ns:m.01xzdz* for the Pantheon. Readable names are only available through an extra join with the *ns:type.object.name* predicate and names do not uniquely identify an entity.
2. Many relations are not binary in nature. Think of a relation like *won-award*. It could associate winners with awards. However, there is a lot of additional useful information like the year when the award was given or the title of the winning work. As this is hard to express in triples, Freebase therefore introduces so-called *mediator objects*. In our example, *winner*, *award*, *year*, etc. all connect to the mediator object with predicates according to their role. This is very useful for relations that just do not have a binary nature (an extreme example is *nutrients per 100g* which only makes sense if at least a food, a nutrient, and an amount are connected). However, Freebase takes this idea very far and even uses mediators for relations like *sibling* or *member of*. Below, we demonstrate how queries quickly become very complicated that way.

3. Freebase also contains information that is not particularly useful for our kind of search. For example, some relations are simply duplicated under another name (e.g., *rdfs:label* duplicates *ns:type.object.name*).

These three points are all very problematic for a user interface like the one for Broccoli. Therefore, we have adapted Freebase to our needs, made the simplified KB publicly available and published our work in (Bast et al., 2014a). The process is algorithmic and thus automated to go far beyond what we could have done for a manual adaption.

Let us compare the SPARQL version of the query *siblings of the Beatles* over both KBs:

Query 4: Siblings of Beatles

On Freebase:

```
PREFIX ns: <http://rdf.freebase.com/ns/>
SELECT DISTINCT ?personname WHERE {
  ?beatlesId ns:type.object.name "The Beatles"@en .
  ?mem ns:music.group_membership.group ?beatlesId .
  ?mem ns:music.group_membership.member ?beatle .
  ?beatle ns:people.person.sibling_s ?sib .
  ?person ns:people.person.sibling_s ?sib .
  ?person ns:type.object.name ?personname
}
```

On FreebaseEasy:

```
SELECT ?sibling WHERE {
  ?beatle <Member_of> <The_Beatles> .
  ?sibling <Sibling> ?beatle .
}
```

Query 4 shows the difference between the formulations of the same query over the original Freebase KB and over our derivation, FreebaseEasy. For Freebase, the query has to use extra *type.object.name* predicates to deal with otherwise unreadable machine IDs for the result entities and the entity *The Beatles* and it has to deal with mediator objects for the *sibling* and *group_membership* relations. In contrast, the second query is much easier to understand and much better to build in an interactive fashion like in the Broccoli UI. We acknowledge that some of the information in Freebase is lost that way, but the overall user experience is increased immensely.

The major challenges to derive FreebaseEasy are automatically resolving mediator objects and finding expressive, canonical names to use as entity IDs. Additionally, we compute a score for each entity. The scores are mostly based on the FACC corpus by Gabrilovich, Ringgaard, and Subramanya, (2013) that recognized and linked Freebase entities in the

Clueweb12 (ClueWeb, 2012) corpus. We describe the exact process in our publication (Bast et al., 2014a).

4.1.4 Experimental Results

We have evaluated both, efficiency and result quality of our system. Efficiency experiments can be found in (Bast and Buchhold, 2013) and on much larger data (ten times as much text and a KB larger by several orders of magnitude) in our more recent publication (Bast and Buchhold, 2017). We also summarize the results in Section 4.2.4. In this section, we present the results of our quality evaluation (Bast, Buchhold, and Haussmann, 2017).

As data we use all text from the English Wikipedia, obtained via `download.wikimedia.org` in January 2013 and the original YAGO (Suchanek, Kasneci, and Weikum, 2007) knowledge base. We are particularly interested in the strength of our *occurs-with* predicate, rather than in evaluating the quality of a particular knowledge base. Thus the small size of YAGO and the incompleteness of its relations is not an issue.

We have evaluated our search on three datasets. Two of these query benchmarks are from past entity search competitions, described in Section 4.1.2: the Yahoo SemSearch 2011 List Search Track (Blanco et al., 2011), and the TREC 2009 Entity Track (Balog et al., 2009). The third query benchmark is based on a random selection of ten Wikipedia *List of ...* pages. Wikipedia lists are manually compiled by humans, but actually they are answers to the same kind of semantic queries that Broccoli answers.

The first two benchmarks use keyword queries (e.g., *astronauts who walked on the moon* or *siblings of Nicole Kidman*). Wikipedia lists have a title that resembles such a keyword query (*List of unicorn startup companies*). For all of them, we have manually generated queries using a target type¹³ (e.g. *Astronaut* or the more general *Person*) and the *occurs-with* predicate to specify co occurrence with words (e.g. *walked moon*) and/or entities (e.g. *sibling <Nicole_Kidman>*). We have relied on the interactive query suggestions of the user interface of Broccoli, but did not fine-tune queries towards the results.

Table 1 shows the impact of scope of text records and thus of our natural language processing. Respectively, our *occurs-with* predicate requires matches within sections, sentences, and the semantic contexts derived as described in Section 4.1.3.

We regard set-related and ranking-related measures. In the following, we briefly describe how the measures are calculated. This description has been written at the same time in the original publication (Bast, Buchhold, and Haussmann, 2017) from which the table is taken, and therefore is replicated into this document:

Our set-related measures include the numbers of false-positives (#FP) and false-negatives (#FN). We calculate the precision (Prec.) as the percentage of retrieved relevant entities among all retrieved entities and the recall as the percentage of retrieved relevant entities

¹³called *class* in the screenshots in Figures 1 and 5

		F1	R-Prec	MAP	nDCG
SemSearch	sections	0.09	0.32	0.42	0.44
	sentences	0.35	0.32	0.29	0.49
	contexts	0.43†	0.52	0.45	0.48
TREC	sections	0.08	0.29	0.29	0.33
	sentences	0.37	0.62	0.46	0.52
	contexts	0.46*	0.62	0.46	0.55
WP lists	sections	0.21	0.38	0.33	0.41
	sentences	0.58	0.65	0.59	0.68
	contexts	0.64*	0.70	0.57	0.69

Table 1: **Performance of Broccoli on the three benchmarks SemSearch, TREC, and Wikipedia lists when running on sections, sentences or contexts. Adapted from (Bast, Buchhold, and Haussmann, 2017). The * and † denote a p-value of < 0.02 and < 0.003 , respectively, for the two-tailed t-test compared to the figures for sentences.**

among all relevant entities. We calculate the F-measure (F1) as the harmonic mean of precision and recall.

For our ranking-related measures, we simply ordered entities by the number of matching segments. R-precision (R-Prec), mean average precision (MAP), and normalized discounted cumulative gain (nDCG) are then calculated as follows: Let $P@k$ be the percentage of relevant documents among the top- k entities returned for a query. R-precision is then defined as $P@R$, where R is the total number of relevant entities for the query. The average precision is the average over all $P@i$, where i are the positions of all relevant entities in the result list. For relevant entities that were not returned, a precision with value 0 is used for inclusion in the average. We calculate the discounted cumulative gain (DCG) as:

$$DCG = \sum_{i=1}^{\#rel} \frac{rel(i)}{\log_2(1+i)}$$

where $rel(i)$ is the relevance of the entity at position i in the result list. Usually, the measure supports different levels of relevance, but we only distinguish 1 and 0 in our benchmarks. The nDCG is the DCG normalized by the score for a perfect DCG. Thus, we divide the actual DCG by the maximum possible DCG for which we can simply take all $rel(i) = 1$.

We observe that there is a significant improvement in F-measure when using semantic contexts over sentences. However, for the ranking-based measures this advantage diminishes. In our publication (Bast, Buchhold, and Haussmann, 2017), we examine this in more detail and show how the semantic contexts are particularly helpful to filter out false positive results.

It is not easy to compare Broccoli against the systems that participated in the original competitions from which our queries are taken. First of all, the competitions have already been completed and, unfortunately, there is no perfect ground truth available for them. This is the case, because their judgments are based on pooling: All participants submit their results and human judges only rate the relevance of what is in that pool. This works well for the challenge but does not guarantee that a complete ground truth is produced. If an entity has not been returned by any competing approach, it remains without judgment – even if it is actually a highly relevant result to the query. Still, there is no better benchmark available and we compare Broccoli to other participants of the TREC Entity Track. At first, directly against the pooling-based judgments (assuming anything not judged is not relevant) but then also against a ground truth extended by our own judgments for hits returned by Broccoli.

	P@10	R-Prec	MAP	nDCG
TREC Entity Track, best	0.45	0.55	n/a	0.22
Broccoli, orig	0.58	0.62	0.46	0.55
Broccoli, orig + miss	0.79	0.77	0.62	0.70
Broccoli, orig + miss + corr	0.94	0.92	0.85	0.87

Table 2: **Quality measures for the TREC benchmark for the *original* ground truth, with *missing* relevant entities, and with errors from categories FP and FN 3,4,5 *corrected*. Adapted from (Bast, Buchhold, and Haussmann, 2017).**

In Table 2 we distinguish three runs. The run against the pooling-based judgments without any modifications is marked as *orig*. The run for which we retrospectively labeled our results that were not included in the original pool is marked as *miss* (for missing judgments). This is the run that we would consider the fairest and thus most meaningful comparison. Finally, we have performed a manual in-depth error analysis in our publication (Bast, Buchhold, and Haussmann, 2017). If we assume perfect auxiliary components (i.e. perfect entity recognition, a perfectly precise and complete knowledge base, and a perfect syntactic parse that serves as input for our natural language processing), we can assess the theoretical potential of the search paradigm. We mark this hypothetical run with all auxiliary errors corrected as *corr*.

Compared to the best performing run at TREC, our results are very strong. Note again, that we consider *miss* the fairest comparison and here the difference, e.g., of 55% to 77% w.r.t R-Prec, is huge. However, we have to keep in mind that the conditions are significantly different. Broccoli answers structured queries and, even if they were not tuned towards the results, considerable human brainpower went into translating the keyword descriptions into such queries.

Thus, for a truly fair comparison another component is needed that translates keyword queries (or even better: natural language questions) into queries for Broccoli. Such a component is another potential source of error. At the same time, our evaluation shows

that such a component may actually be well worth building. If it works decently, the system can be expected to beat the competition. If it would work perfectly, and other auxiliary components would do so as well, the *(orig + miss + corr)* column in Table 2 promises phenomenal potential.

4.2 Efficient Indexing and Query Processing

Our work on efficient indexing and query processing is a crucial aspect of all of our systems and comprises most of the technical contributions presented in this thesis. Our novel algorithms and data structures for SPARQL+Text search are far more efficient than related approaches.

There are two key publications: The first (Bast and Buchhold, 2013) describes the index behind Broccoli that makes interactive queries of this kind possible. QLever (Bast and Buchhold, 2017) builds upon this but integrates the functionality as an extension to the SPARQL language. Hence, it is able to answer all queries that are possible in Broccoli, but also more than this by providing full¹⁴ SPARQL support. Therefore, it requires a more sophisticated index for the KB part and an advanced query planner that has to be able to deal with our extensions for text search. On top of that, QLever makes significant improvements to the text index that backs up Broccoli. In this section, we focus on the most recent and furthest developed version of our indexing and query processing and thus more on QLever than on Broccoli.

4.2.1 Problem Statement

We want to create an efficient query engine for SPARQL+Text queries on very large data. Recall that the query language is SPARQL extended by two special predicates: *ql:contains-word*, which allows words and prefixes to be linked to text records, and *ql:contains-entity*, which allows this linking for entities and variables. Query results can be ranked by the quality of the text match and matching text snippets can be returned as part of the result.

We want to match or improve upon the speed of state-of-the-art SPARQL engines for pure SPARQL queries and to significantly outperform it on SPARQL+Text queries. Our data structures and algorithms should handle arbitrary knowledge bases and text corpora.

4.2.2 Related Work

In Section 2, we have discussed how similar search paradigms relate to ours. We have identified two areas with systems that are closely related and aim for efficiency for comparable kinds of queries: (1) SPARQL engines, that partially cover our query language by nature and for which we can emulate the full spectrum of our queries (see Section 2.2) and (2) systems for (semi-) structured queries on combined data that answer similar queries but usually return documents rather than entities (or tuples). Here, we take a closer look at the data structures and algorithms used by those systems for each of the

¹⁴QLever provides full SPARQL support in terms of the core of the language. Some advanced SPARQL features still have to be added to QLever but we are not aware of anything that poses general problems for our system and index architecture.

two categories and summarize their basic techniques. We want to remark that there is some overlap with our discussion of related work in the publication of QLever (Bast and Buchhold, 2017) which has been written around the same time.

SPARQL Engines

As described by Elliott et al., (2009), SPARQL queries can be rewritten to SQL and all the big relational databases now also provide support for SPARQL. In contrast, there are also systems, often called triple stores, whose index and query processing are specifically tailored towards SPARQL queries over knowledge-base data.

A fundamental idea for tailor-made indices for SPARQL engines is to index all possible permutations of the triples. With triples consisting of subject (S), predicate (P) and object (O) this leads to 6 (SPO, SOP, PSO, POS, OSP, OPS) permutations in total. This idea was first published for Hexastore (Weiss, Karras, and Bernstein, 2008) and RDF-3X (Neumann and Weikum, 2008). Our engine, QLever, also makes use of this idea. We want to remark that the two permutations PSO and POS suffice for many semantic queries. In fact, all queries that do not use variables for predicates (and thus all queries that can be asked in the Broccoli search engine) are supported with only those two permutations.

In the following, we take a closer look at two systems: (1) RDF-3X, one of the original systems that uses 6 permutations and that inspired several aspects of the knowledge-base side of QLever and (2) Virtuoso¹⁵, a commercial product (with an open-source version) that is widely used in practice, e.g., for the public endpoint¹⁶ for the DBPedia KB (Auer et al., 2007), and in many SPARQL performance evaluations. Virtuoso builds full indices for only the PSO and POS permutations, as described below.

The main idea behind RDF-3X is to index all six permutations of the triples as described above. Queries then make sure to use the optimal permutation for each scan and thus many join operations can be implemented as merge joins without explicitly sorting the inputs before. Inspired by this work, we also follow the same general idea in our system QLever. However, within each permutation we rely on our own data layout to further optimize the speed at which scan operations can be executed. Query execution in 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); see (Neumann and Weikum, 2009). SIP allows multiple scans or joins that operate on common columns to exchange information about which segments in these columns can be skipped. QLever forgoes pipelining and SIP in favor of highly optimized basic operations and caching of sub-results.

Virtuoso is built on top of its own full-featured relational database and provides both, a SQL and a SPARQL front-end. There is no research paper but a very insightful article is

¹⁵<https://virtuoso.openlinksw.com/>

¹⁶<https://dbpedia.org/sparql>

available online¹⁷. Virtuoso builds PSO and POS permutations¹⁸ and additional partial indices (SP, OP) to deal with variable predicates, albeit less efficiently than with the more frequent variables subjects and objects. The partial indices cannot answer queries on their own. For a triple pattern with variable predicate, they yield SP or OP (depending on whether subject or object are given) pairs which can be used to access one of the two full permutations (e.g., by sorting the matching pairs by P). If variable predicates are very important for a particular application, the user can decide to also build full indices for other permutations, thus trading index size for efficiency on that particular kind of query. Since Version 7, the triples inside a permutation are stored column-wise.

In Section 2.2 we have mentioned that Virtuoso supports full-text search via its *bif:contains* predicate and argued how this extension is less powerful than our extensions to SPARQL but can be used to emulate them – with low efficiency, though. The functionality is realized via a standard inverted index and allows query keywords to match literals from the knowledge base. The approach is typical for keyword-search support in SPARQL engines and also taken by Jena (see <http://jena.apache.org/documentation/query/text-query.html>) and BlazeGraph (see <http://wiki.blazegraph.com/wiki/index.php/FullTextSearch>). We want to remark again, that this does not support entity occurrences anywhere in the text. Therefore these extensions are less powerful than QLever’s and do not offer anything comparable to its *ql:contains-entity* predicate.

Systems for Queries on Combined Data

KIM (Popov et al., 2004) was the first system for combined search on a knowledge base linked with a text corpus. KIM is based on a standard inverted index (and on off-the-shelf search engine software) and builds inverted lists for knowledge-base entities. These lists then contain the document IDs for an entity’s occurrences in the text. Thus, entities are treated just like normal words. SPARQL queries are issued to a separate engine (again off-the-shelf). Then a keyword query is constructed as conjunction of the keywords and a disjunction of all result entities from the SPARQL query. This query is then issued to the text search engine. The final query results are documents, not entities. Obviously, this approach does not allow to arbitrarily mix and nest KB and text parts in the query. As a more important drawback, it also becomes very inefficient when the result of the SPARQL query is large and thus a very large disjunctive text query is build and has to be processed by the text search engine.

Mimir (Tablan et al., 2015), which can be considered KIM’s successor, tries to overcome the efficiency issue by adding more artificial terms to the index. These terms represent entire classes of entities, e.g., there can be an inverted list for all entities of type *person*, the more specific type *politician*, or according to our example for a category like *buildings in Europe*. The natural limitation to this approach is, that one can only index a certain amount of such lists. Available lists can be chosen to represent the categories of entities,

¹⁷ <http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning>

¹⁸ Actually, Virtuoso stores quads instead of triples with an additional *graph* attribute and thus stores PSOG and POGS permutations, but for the purpose of this explanation it does not matter.

that are most frequently used in queries, but they can never cover everything interesting selectable via SPARQL. For arbitrary SPARQL queries, Mimir falls back to the same inefficient approach as KIM.

ESTER (Bast et al., 2007) overcomes this problem. Unlike KIM and Mimir it does not use a standalone (and off-the-shelf) SPARQL engine, but entities and their relations are represented in artificial text documents, that are indexed in addition to conventional text documents. The search yields 4-tuples (doc ID, word ID, position, score). SPARQL-like queries can then be answered by a mix of positional and prefix search operations and, in addition to well-known intersect operation on doc IDs, lists may also be re-sorted and intersected on word IDs. However, like KIM and Mimir, search results are text documents and not entities. Thus, none of these approaches is suited for processing general SPARQL queries, which are entity-centric.

All three systems discussed above share some characteristics: Some semantic queries can be very fast if they touch moderate numbers of entities, and especially Mimir and ESTER benefit if sufficiently specific types (e.g., *building* rather than *person*) are used. However, it is often possible to find queries that take very long to process. For Mimir those are queries that involve a complex SPARQL part that still return many entities and for ESTER the use of very unspecific types (high up in a hierarchy) and similar relations can lead to very large lists that then have to be sorted to order them by word ID so that they can be joined/intersected on that attribute. Finally, neither of those systems provides true SPARQL support: the document centric approaches do not return a list of entities, let alone tuples of multiple matching variables.

4.2.3 Approach

There are two cornerstones that make our systems efficient: indexing and query processing. For the indexing, we have developed two data structures: a knowledge-base index and a text index. Both indices are designed so that the data needed at any step during query processing is stored contiguously and without any extra data in between. We achieve this by introducing some redundancy. The query processing also consists of two important parts. First, efficient execution trees have to be found. Therefore, we use a dynamic programming algorithm in which we have to account for our special operations for text search. Only after the optimal execution tree is found, we process the query and make use of our index data structures to efficiently compute its results. In the following, we describe the layout of our index data structures and the algorithms used for query processing.

Knowledge-Base Index

The first pillar of our indexing is the knowledge-base index. Our system QLever (Bast and Buchhold, 2017), makes full use of this data structure. The index behind Broccoli, as described in (Bast and Buchhold, 2013), only uses the text index as explained later (actually a slightly inferior predecessor of what is described for QLever and in this docu-

ment), and a simplistic knowledge-base index. This simple KB index just keeps two lists of pairs of subject and object IDs; one list sorted by subject and the other one by object.

The knowledge-base index of QLever is more advanced. Like RDF-3X (Neumann and Weikum, 2010) and Hexastore (Weiss, Karras, and Bernstein, 2008), we first sort the triples (S = subject, P = predicate, O = object) in all possible ways and create six (SPO, SOP, PSO, POS, OSP, OPS) permutations. For each permutation, we build multiple lists of binary data. These lists are different from what is used by other systems. In the following, we examine a PSO permutation for a *Film_Performance* predicate as an example. Other predicates and the five other permutations are handled in the same way. Note that the user may choose to build all six or only two (PSO and POS) permutations¹⁹.

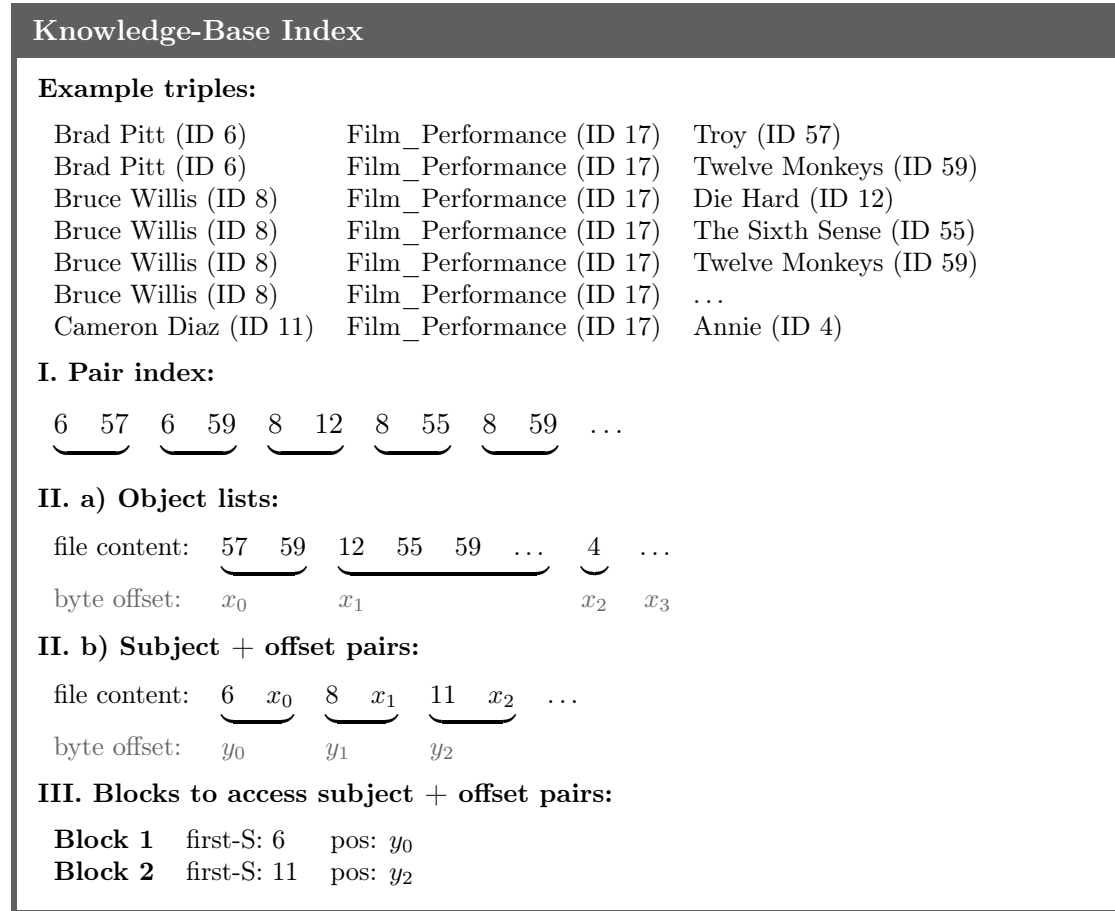


Figure 7: The components of our knowledge-base index, exemplified for the PSO permutation for a *Film Performance* predicate. Originally published in (Bast and Buchhold, 2017).

¹⁹If an application, like Broccoli, does not need to support variables for predicates, using only two permutations is enough to answer all other queries.

For each predicate in the permutation, we build the binary lists shown in Figure 7. The pair index (I.) is used when the full list for the predicate is needed during query processing. This happens for SPARQL patterns like *?s <Film_Performance> ?o*. The object lists (II.a) are used when we only need the objects, e.g., *<Brad_Pitt> <Film_Performance> ?o*. Note that if we only needed subjects or predicates, we would use a different permutation. The subject+offset pairs (II.b) and blocks (III.) are used to efficiently find the byte offsets between which we have to read to get such a list of objects. For special cases like "functional" predicates (only one object per subject) and predicates with only few triples, the offsets can also point directly into the pair index. We explain this in detail in our publication (Bast and Buchhold, 2017).

Text Index

Text Index

Example text:

Text Record 123:

The Augustan<Augustan_Age> Pantheon<Pantheon> was destroyed in a fire.

Text Record 234:

If a boiler<Boiler> is dry-fired it can cause catastrophic failure.

Text Record 520:

The Pantheon<Pantheon>, originally built by Agrippa<Marcus_Vipsanius_Agrippa> but destroyed by fire in 80, was rebuilt under Hadrian<Hadrian>.

Inverted lists for prefix fire*:

I. Word part (IDs: fire:17; fired:20):

Record IDs: ... 123 234 520 ...

Word IDs: ... 17 20 17 ...

Scores: ... 1 1 1 ...

II. Entity part (IDs: Augustan_age:12; Boiler:15; Hadrian:43; Marcus_Vipsanius_Agrippa:52; Pantheon:60):

Record IDs: ... 123 123 234 520 520 520 ...

Entity IDs: ... 12 60 15 43 52 60 ...

Scores: ... 1 1 1 1 1 1 ...

Figure 8: The components of our text index, illustrated for an example text; adapted from (Bast and Buchhold, 2017).

The second pillar of the indexing behind QLever is its text index. It is based on a classic inverted index which stores, for each term, the sorted list of IDs of all documents (or text

records) it occurs in. These lists of record IDs can be extended to lists of postings, that also contain additional items, e.g., scores or positional information. On top of that, Bast and Weber, (2006) have shown how word IDs can be included to allow inverted lists per prefix instead of per term with very little overhead, enabling highly efficient prefix and faceted search. Inspired by this work, we also follow this approach and store word IDs. This obviously gives us the opportunity to efficiently answer prefix queries, but more importantly allows storing entity IDs (as word IDs) within our inverted lists.

We then augment our inverted lists by postings for all entities that co-occur in one of the contained text records. Initially in (Bast and Buchhold, 2013), we interleaved such entity postings in the inverted lists for each index term, but in (Bast and Buchhold, 2017) we have shown that using separate lists has many benefits. These separate lists are depicted as word part (I.) and entity part (II.) in Figure 8. Intuitively, one can think of this entity part as pre-computation for all queries of the kind: *all entities that occur with <word>*, except that we do not yet organize results by entity and do not yet aggregate their frequencies. Instead, each entity will occur multiple times in an inverted list, if it occurs in multiple corresponding text records.

All lists are stored compressed. We gap-encode record IDs and frequency-encode word IDs and scores. Then we use the Simple8b (Anh and Moffat, 2010) algorithm to compress the resulting lists of small integers.

Note that each text record can be included in several inverted lists and thus an entity posting may be stored multiple times in our text index. The blowup factor induced by this redundancy is determined by the average number of entities per text record. In our experiments in (Bast and Buchhold, 2013) we have found this factor to be around 2, which is tolerable. More importantly though, the latest version of our index alleviates the problem further: The lists touched for each query are exactly what is needed to answer the query; see (Bast and Buchhold, 2017), Section 4, for a detailed description of how this is achieved. The blowup only impacts the size of the index on disk and not the size of posting lists to read and traverse. As a byproduct of this, pure keyword queries, that do not involve entities, are processed exactly like they were in a classic inverted index. This is another advantage over Broccoli where the blowup factor affected all inverted lists and thus caused the system to be slower on pure text queries (see Table 3 in Section 4.2.4).

Query Processing

For Broccoli, query processing is straightforward. Its user interface and query language enforce that queries have a tree structure. We simply process these trees in a bottom-up fashion and cache the results of subtrees for reuse. Each of these sub-results is simply a list of entities. For snippet generation, we use a top-down approach and retrieve snippets and other additional information to display for exactly those top-ranked results that are shown in the UI. Note that the computations made by this top-down approach would be prohibitively expensive to yield all tuples as returned by a SPARQL query. However,

reconstruction for the few hits displayed in the UI takes negligible time. Details on the query processing in Broccoli can be found in (Bast and Buchhold, 2013).

The query processing of QLever is more intricate. Support for the full SPARQL language means that we also have to (1) allow cyclic queries, (2) allow variables for predicates, and (3) produce result tables, i.e. lists of tuples, not single entities. Neither of these three requirements is fulfilled by the simplistic query processing in Broccoli. On top of that, we want to be able to process all queries in the ideal way that leads to the fastest execution time. We have found that some queries can be processed much faster when planned properly compared to just solving Broccoli’s user-made queries from bottom to top. QLever’s query processing thus has two parts: query planning and query execution.

A standard procedure for processing SPARQL queries is, just like for SQL queries, to build execution trees that have operations as their nodes. Operations that do not take any sub-results as their inputs (e.g., scans for index lists) become leaves, operations that require one or more sub-results as their input (e.g., SORT or JOIN operations), become the intermediate nodes. These trees are then processed in a bottom-up fashion to compute the result to a query. For SPARQL queries, each triple pattern in the WHERE clause of the query corresponds to a scan, each shared variable between triples corresponds to a join. For QLever, this is a bit more complex: Its special-purpose predicates, *ql:contains-entity* and *ql:contains-word*, do not each correspond to scanning an index list. Instead, QLever’s special text index allows processing all triples pertaining to the same text record variable in one go.

In the following, we look at the example query from the introduction (*buildings in Europe that were destroyed in a fire* with its SPARQL+Text representation given as Query 2) and how it is processed by QLever. For a more complex example with co-occurrence between multiple entities, we refer the reader to (Bast and Buchhold, 2017).

Before we can construct the execution tree, we first interpret the SPARQL+Text query as a graph. Every triple pattern from the WHERE clause of the query corresponds to a node in this graph. If a variable is shared between two or more patterns, we draw an edge between their corresponding nodes. This is a standard approach and also taken by RDF-3X (Neumann and Weikum, 2010). However, we have to account for the two QLever-specific predicates and our operations for text search.

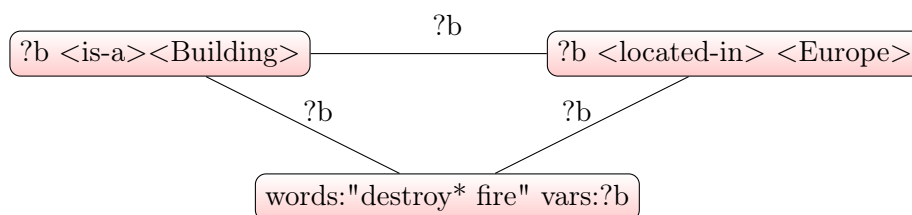


Figure 9: Graph for Query 2 from Section 1 (*buildings in Europe that were destroyed in a fire*). Cliques formed by a shared variable for a text record are collapsed into a single node.

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 9 where the bottom node covers two triple patterns from the original query (in particular, *?t ql:contains-word "destroy fire"* and *?t ql:contains-entity ?b*).

As a next step, we build execution trees from this graph. Inspired by the query planning for RDF-3X, we use an approach based on dynamic programming. This is practice-proven and has been studied well for relational databases (see (Moerkotte and Neumann, 2006) for an overview).

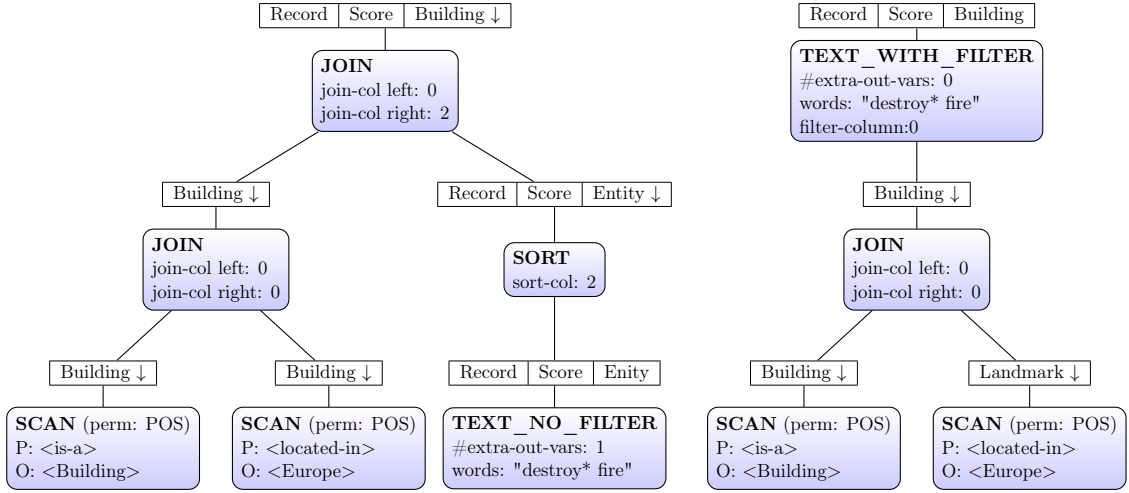


Figure 10: **Two (out of many possible) example execution trees for the query from Figure 9. The right one is smaller, because of the complex *Text with Filter* operation.**

In the graph from Figure 9, every node corresponds to an index operation and every edge corresponds to a possible join. Figure 10 depicts two (out of many possible) plans for the example query. We find the optimal execution tree by starting with leaves. Leaves directly correspond to nodes in the graph – there is a leaf in the execution tree for every node in the query graph and vice-versa.²⁰ Then we iteratively connect two of these subtrees by adding a join operation to create larger and larger trees. Since we use merge joins, a sub-result, that is supposed to become an operand of a join, may have to be prepared by a sort operation first. We add these sort operations whenever needed. In this way, we enumerate all possible subtrees that cover two, three, etc. nodes of the query graph. In the end, we want all leaves to be connected. At each step we compute cost and size estimates for the intermediate execution trees and their results. We prune

²⁰The special *Text with Filter* operation is an exception to the rule that nodes in the query graph directly translate to leaves in the execution tree, but for the purpose of the explanation here, this special case is not important.

away all execution trees that are surely inferior to others. For details on this process and especially on our own estimators, we refer the reader to (Bast and Buchhold, 2017).

Both example execution trees in Figure 10 make use of one of the special text operations available in QLever. The text operation in the left execution tree simply yields co-occurring entities for the text part. Therefore it accesses the text index, and then arranges matches by entity. Afterwards, joins are performed to arrive at the final result. The text operation in the right execution tree takes a sub-result as input, here the list of buildings in Europe, and filters the postings from the text index so that fewer matching elements have to be arranged by entity. For the intricacies of our text operations, especially when multiple entities are supposed to co-occur with each other, and the algorithms to efficiently compute the result with the help of our text index, we refer the reader to (Bast and Buchhold, 2017).

4.2.4 Experimental Results

In (Bast and Buchhold, 2013), we have evaluated the special-purpose index behind Broccoli against two possible ways of using an inverted index to answer similar queries. (1) A classic inverted index with additional inverted lists for each type (e.g., *building*) in the KB. Such a list contains all occurrences of entities of that type. In Table 3, we use *Inv* to refer to this approach. (2) A classic inverted index with an additional forward index (a mapping from record ID to all entities that occur within the record). In Table 3, we use *Map* to refer to this approach.

Unfortunately, these experiments predate our work on QLever and thus QLever is not included in the comparison. However, we summarize our more recent evaluation later in this section which compares QLever to Broccoli (among state-of-the-art SPARQL engines). When considered together, the two experiments also provide insight on how QLever compares to the approaches based on a classic inverted index.

Our first experiments from (Bast and Buchhold, 2013) are conducted on all the text of the English Wikipedia (from January 2012) and the original YAGO KB (Suchanek, Kasneci, and Weikum, 2007). The Wikipedia text amounts to a 40 GB XML dump with 2.4 billion word occurrences, 285 million recognized entity occurrences, and 334 million sentences which we decompose into 418 million text records with the natural language processing used by Broccoli.

In the original publication, we have generated 8000 synthetic queries from 8 categories. Here, we use a less fine-grained breakdown and report averages over 2000 pure text queries and 5000 SPARQL+Text queries (more precisely: the subset of SPARQL+Text supported by Broccoli as described in Section 4.1). For a detailed breakdown and in-depth evaluation, we refer the reader to the publication itself.

The results in Table 3 are not surprising: For pure text search, the classic inverted index is better than Broccoli because of the additional items inserted into the inverted lists by Broccoli. As explained in Section 4.2.3, the more recent QLever is identical to *Inv* here,

	Inv(m)	Map(m)	Map(d)	Broccoli(m)	Broccoli(d)
Text	21ms	26ms	44ms	57ms	107ms
SPARQL+Text	898ms	197ms	54s	74ms	148ms
Index Size	16GB	11GB	11GB	14GB	14GB

Table 3: **Comparison of Broccoli with other indexing strategies based on a classic inverted index. We report average query times. Runs are marked with (m) for runs with all index data in memory (in the file system cache of the operating system, to be precise) and with (d) for runs where index data has to be read from disk. The table is based on the results reported in Table 1 of our publication (Bast and Buchhold, 2013).**

because all additional items are located within extra lists (to be precise, as the *entity part* in Figure 8 from Section 4.2.3) and only read when needed. The benefit is demonstrated later in Table 4 where we compare QLever against Broccoli. For SPARQL+Text queries, Broccoli performs much better than the baselines, though. *Map* actually comes fairly close if all index data is in memory but it is still significantly slower than Broccoli. With cold caches and index data read from disk, *Map* is incredibly slow. This is as expected, because for each of the many involved text records, it has to read a list from a random location on disk.

In (Bast and Buchhold, 2017), we have compared both of our systems against two state-of-the-art SPARQL engines which we have already described in Section 4.2.2. To emulate SPARQL+Text search, we make use of strategies explained in Section 2.2. We compare QLever and Broccoli against RDF-3X using emulation strategy I from Figure 2, and Virtuoso with *bif:contains*, its special-purpose predicate for full-text search, using emulation strategy III.

In (Bast and Buchhold, 2017) we used queries from 12 categories. Each category contains 10 manually crafted queries with a clear narrative (like the examples used throughout this document). We treat queries involving a prefix in separate categories in order to avoid distortion of averages. This is necessary because of two reasons: They are not supported by RDF-3X and they are significantly slower than other queries in Virtuoso.

The categories labeled *SemSearch* are translations of queries of the Yahoo SemSearch 2011 List Search Track (Blanco et al., 2011) and contain more than 10 queries. While we used prefix search to formulate 10 of these queries, because it helps a lot for accurately capturing the query intend, 49 queries only require conventional words. We mark the prefix and word variants by P and W suffixes.

The categories *One Scan* and *One Join* contain simple queries that are directly answered by a single scan and by two scans and one join operation, respectively. However, some queries touch very large predicates and thus these “easy” queries can actually take considerable time to process. Categories called *Is-a + Text* and *Is-a + Prefix* search for

occurrences of entities of a given type together with something matching in the text. Queries from this class are very typical semantic queries. For example, queries for *astronauts who walked on the moon* or for *plants with edible leaves* are formulated in this way.

The example from the introduction, *buildings in europe that were destroyed in a fire*, also contains an extra predicate (*located in europe*) to match in the KB. Therefore, we would have included it in the *Complex Mixed* category (albeit as one of the smallest queries alongside others with significantly more query triples and complexity).

We perform experiments on two datasets and report the main results for both of them: FreebaseEasy+Wikipedia and Freebase+ClueWeb. Each of them consist of a knowledge base and a text corpus in which the entities have been linked.

FreebaseEasy+Wikipedia

This dataset consists of the text of all Wikipedia articles from July 2016 linked to our FreebaseEasy knowledge base (Bast et al., 2014a). Entities have been linked to their occurrences by the pipeline we use for Broccoli as described in Section 4.1.3. The KB has 362 million triples, the Wikipedia text corpus has 3.8 billion word and 494 million entity occurrences.

Table 4 lists the average query times for each of the 12 categories as well as the size of the index on disk and the amount of memory that was used. Note that we report the maximum memory used by the process. All runs started with an empty disk cache (we explicitly cleared the disk cache of the operating system) but it is possible that it has been used during the run. QLever is fastest across all categories and produces the shortest query time for 89% of all individual queries.

The results are not surprising for SPARQL+Text queries, for which our systems QLever and Broccoli were explicitly designed. However, it may come as a surprise that QLever also performs best in other categories contain pure SPARQL queries. Most notably, there is a large difference for the *Complex SPARQL* set. There are a couple of reasons for that. The most obvious one is reflected in the index size without text, especially compared to Virtuoso. We deliberately add redundancy in our knowledge-base index for the sake of fast query times. The reason for this choice is that we are more interested in the *total* index size, including the text index. Here, the size of knowledge-base data becomes less and less relevant and effective compression of the text index is much more important. Right now, we do not even compress the binary lists of our knowledge-base index. While we may change this in the future, we will definitely choose a compression algorithm that is optimized for speed, rather than space. Another possible reason for QLever’s strength on the pure SPARQL queries is that our emulations insert many triples into the KB. Thus, other systems effectively search a larger KB. Ideally, their indices should ensure there this data is not touched for a pure SPARQL query and thus that there is no significant impact on performance, but we cannot guarantee that. Finally, there are some advanced features of Virtuoso and (partially also of) RDF-3X that have yet to be added

to QLever: insert and update operations, the ability to run in a distributed environment, and so on. While we do not see principle problems and why the addition of those had to significantly harm the performance of QLever, we also have to remark that the larger palette of supported features may put Virtuoso at a disadvantage when comparing on the kinds of queries QLever was made for.

	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	12.1 s	15.0 s	427 ms	191 ms
Is-a + Word	1738 ms	941 ms	178 ms	78 ms
Is-a + Prefix	-	20.5 s	310 ms	118 ms
SemSearch W	1078 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 ²¹
Index w/o Text	17 GB	9 GB	8 GB ²²	49 GB ²³
Memory Used ²⁴	30 GB	45 GB	10 GB	7 GB

Table 4: **Average query times for queries from 12 categories on FreebaseEasy+Wikipedia. Originally published in (Bast and Buchhold, 2017). All caches were cleaned before each run and index data had to be read from disk. Similar to runs marked with (d) in Table 3.**

Compared to RDF-3X, there is another trade-off we make. QLever keeps a significant amount of metadata, and its entire vocabulary (an array of string items whose index corresponds to the items ID used in the index) in RAM at all times. Thus it (and Virtuoso

²¹The 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.

²²only supports two permutations and limited features

²³20 GB for the permutations that are really needed

²⁴All 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.

as well) requires several seconds²⁵ to start. RDF-3X, in contrast, has all relevant data stored on disk. Even if this data is intelligently aligned, it still puts RDF-3X at a slight disadvantage compared to the other two systems, especially when output for large results has to be produced and string representations of these results have to be restored.

Freebase+ClueWeb

This dataset is based on FACC (Gabrilovich, Ringgaard, and Subramanya, 2013), which is a combination of Freebase (Bollacker et al., 2008) and ClueWeb12 (ClueWeb, 2012) where entity occurrences have already been linked to their KB representations by the authors. The KB has 3.1 billion triples, the text corpus has 23.4 billion word and 3.3 billion entity occurrences. These numbers are roughly ten times larger than for the FreebaseEasy+Wikipedia dataset. Unfortunately, we could not successfully load a dataset of this size into the other systems, because they failed to index it 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.

	QLever cold cache	QLever warm cache
Only Text	1279 ms	840 ms
SemSearch W	390 ms	214 ms
SemSearch P	613 ms	339 ms
Complex Mixed	1021 ms	603 ms
Very Large Text	2245 ms	1849 ms

Table 5: **Average query times for QLever for queries from the 5 hardest categories on Freebase+ClueWeb. Adapted from (Bast and Buchhold, 2017).**

Table 5 shows the performance of QLever on the large Freebase+ClueWeb dataset for queries of the 5 hardest, and thus most interesting, categories that involve text search. We did not translate queries from other categories, because manual translation to the schema of Freebase is quite time consuming. The results show that QLever has no problems with scaling to a collection of these dimensions and we are confident that also a text corpus larger by another order of magnitude would not cause any problems. However, we are not aware of a text corpus of that dimension in which entity occurrences have been linked to a KB.

²⁵For the FreebaseEasy+Wikipedia dataset about 15 seconds are needed for startup, for Freebase+Clueweb, startup can take up to a few minutes.

4.3 Relevance Scores for Knowledge-Bases Triples

Our work on Broccoli and SPARQL+Text search has a strong focus on user needs. Efficiency is one major aspect of that, but so are result quality and ranking. In our prototypes, we have identified an issue with ranking entity queries. Consider a SPARQL query for actors²⁶. SPARQL without explicit ORDER BY does not require a particular ordering of the result. With 428,182 actors in Freebase, a random (or lexicographical) ordering of such a result list would most probably start with unknown actors and is obviously not what a user wants to see.

A natural way to rank the result is to compute some form of popularity score (e.g., what we did in (Bast et al., 2014a)) and order by that score. However, this gives us *George W. Bush* as the top result. In Freebase, he is in fact listed as an actor and he even has an IMDb page²⁷ listing his occurrences in movies. Thus, this result is not wrong, strictly speaking. Still, such a ranking clearly does not constitute a proper list of actors as expected by a user. We want to remark that this can also happen when the correctness of triples is even less debatable: Quentin Tarantino is much better known as a director than as an actor, but he undoubtedly has frequent cameo appearances in his own movies. Thus, the task is different from work that deals with inaccurate or conflicting KB triples. To overcome the issue, we need a way to know how strongly an entity belongs to some type. We compute relevance scores that capture exactly this. These scores can then be combined with popularities to produce a sensible ranking.

This also goes the other way round: Think of a query for the professions of someone. For instance, Barack Obama has taught law and worked at a Chicago law firm. He rightfully has the professions *lawyer* and *law professor* in addition to *politician*. But a query for his professions should bring up *politician* first, because that is what he is world-famous for. The easier the query, the more apparently we benefit from such scores. Accordingly, the simple query for actors reveals such a glaring issue. However, the idea behind the scores also applies to complex queries and thus also to SPARQL+Text search. For example, a query for politicians that went on to become leaders of large corporations should rather find people known for being politician like the former German chancellor Gerhard Schröder than former HP CEO Carly Fiorina, who once unsuccessfully ran for the United States Senate but has a very strong signal for being a CEO from a Wikipedia text corpus.

We have found that this problem occurs for all type-like predicates. We have already used examples from a *profession* predicate, but the same thing happens for *nationality* (many people have multiple legal nationalities or had different nationalities throughout their lives), *genre*, ..., and also generic *type* predicates.

We have published a full research paper (Bast, Buchhold, and Haussmann, 2015) where we identify this problem, create a benchmark and introduce several models to compute

²⁶We use a similar example in our publication (Bast, Buchhold, and Haussmann, 2015). We reuse it here, because it makes the problems that we are trying to solve very obvious.

²⁷<http://www.imdb.com/name/nm0124133/>

such scores from a text corpus with linked entity occurrences (the same data model as in all our work discussed in this thesis). Afterwards, we have also organized a task at the 2017 WSDM Cup (Heindorf et al., 2017) where participants were asked to develop algorithms that compute such scores.

4.3.1 Problem Statement

Given a type-like predicate, we want to compute a score for each of its triples, that captures how strongly the entity belongs to that type. Therefore we solve two subproblems: (1) Create a benchmark based on human judgments. This allows us to assess the effectiveness of models for the task. (2) Devise several models to compute such scores and compare them on the benchmark.

4.3.2 Related Work

In general, our work has introduced a novel problem and we are not aware of other efforts to produce the same scores. However, the need for such scores is not entirely new: There are a couple of approaches to ranking knowledge-base queries that assume such or similar scores as given and build ranking models on top of them. For example, Cedeño and Candan, (2011) propose Ranked RDF, an extension to classical knowledge-base data that accounts for scores associated with triples, and Elbassuoni et al., (2009) propose a ranking model for SPARQL queries with possible text extensions based on Language Models. Again, triples are expected to have relevance scores. Neither work discusses methods to obtain high-quality scores for an existing KB.

Our problem is topic modeling where documents are automatically assigned one or more topics. Classically, the topics are not known beforehand, but only a number of topics to infer is given. A prominent approach to topic modeling is Latent Dirichlet Allocation (LDA) by Blei, Ng, and Jordan, (2001). LDA assumes a generative model: For each word in a document, first a topic t is picked with probability $P(t/doc)$ and then a word w is picked with probability $P(w/t)$. The special part is that the topic and word distributions are assumed to have sparse Dirichlet priors. Intuitively, the use of these priors models the assumption that a single document usually deals with only small subset of all topics and that each topic only uses a specific part of the entire vocabulary.

At first glance, LDA may seem quite far from our problem. After all, we are already provided with a very limited set of topics (types, in our case) for each document (entity, in our case). However, an extension called LLDA (Ramage et al., 2009) is related more closely. Originally, LLDA has been intended for retrieving snippets relevant to a specific tag (label) from documents that, as a whole, have been assigned such tags by humans. Queries for these tags can easily match appropriate documents, but it is not clear how to retrieve relevant snippets for the specific tag. This is where LLDA comes in. Therefore, it extends the model of LDA further by integrating the human labels. While we are not interested in finding snippets, we can treat our available types like those human tags so

that the hidden variables for $P(t/doc)$ somewhat reflect the relevance scores we seek for our triples. In the following, we introduce our own, simpler but purpose-built generative model and compare it to LLDA in our experiments; see Section 4.3.4 where we show that it performs much better than LLDA.

4.3.3 Approach

Our approach consist of two steps. First, we create a high-quality benchmark for the novel task, then we develop several models and evaluate them on the benchmark.

Creation of the Benchmark

In order to generate a benchmark, we need many judgments from multiple humans who should ideally not be part of our team and thus make their judgments independently from our intended use case. Therefore, we have set up a crowdsourcing task on Amazon Mechanical Turk. We did this for two type-like predicates: *profession* and *nationality*.

We than designed the experiment in a way such that we gave a person (along with its Wikipedia link) and all of its professions (nationalities, resp.) to each human judge. The judge was asked to label all professions as either primary or secondary. Figure 11 shows one instance of this task.

Person: Barack Obama [Wikipedia page](#)

PRIMARY The person is well-known for this profession or a typical example for people with this profession.

SECONDARY This is no primary profession of the person.

Unlabeled Professions

✚ Politician	i
✚ Lawyer	i
✚ Law Professor	i
✚ Author	i
✚ Writer	i

☒ I only used Wikipedia for my decision. ☐ I used other resources as well.

Figure 11: **Condensed illustration of the crowd sourcing task.** All professions must be dragged into the box for primary or secondary professions. Originally published in (Bast, Buchhold, and Haussmann, 2015).

We gave each instance (person) to seven independent judges. By counting the number of *primary* votes we obtain a score between 0 and 7 for each person-profession combination.

Models to compute scores

In the following we always use the *profession* predicate as an example, but we want to remark that everything is just as valid for other type-like predicates.

Our models are all based on text that we associate with entities. We use the text from Wikipedia with all entity occurrences linked to their representation in the KB: exactly the kind of data the Broccoli search engines operates on. Collecting all sentences (or semantic contexts as described in Section 4.1.3) in which an entity occurs, gives us a virtual document for that entity. We have found this virtual document to provide more useful information than the Wikipedia article of the entity. On top of that, it can be used for entities that do not have Wikipedia pages (if they have been recognized and linked in the corpus). We have also considered models that were only based on knowledge-base data, but found them not to work sufficiently well.

For each profession we could combine the virtual documents of all entities with that profession, but we have found that we get better results if we limit the combination to entities with *only* that one profession (modulo parent types in a type hierarchy). These positive examples can serve as training data to learn models for each profession. It is also possible to generate negative training examples by using text for entities that do not have the profession at all.

In our publication (Bast, Buchhold, and Haussmann, 2015), we discuss several models, variants and combinations. In the following we briefly explain the three most important ones: (1) a binary classifier, (2) a model based on computing weighted indicator words for each profession, and (3) a generative model.

The binary classifier is based on logistic regression. Features are word counts normalized by the total number of word occurrences. We use positive and negative training examples as explained above.

Our weighted indicator words are based on their *tf-idf* values within the virtual document of a profession (the *idf* is calculated across the collection of all virtual entity documents). We rank all words by that value and assign $1/\text{rank}$ as their weight. To make a decision for the professions of a given entity, we go through the virtual text document of that entity and, for each of its professions, sum up the corresponding word weights.

Our generative model is based on the same process as discussed for LDA and LLDA in Section 4.3.2: We assume each document is generated by the following process: Pick a profession with probability $P(\text{prof/doc})$, then pick a word with probability $P(\text{word/prof})$. However, we set the Dirichlet priors aside. Thus, our model is similar to the one underlying pLSI (Hofmann, 1999) with the difference, that we infer $P(\text{word/prof})$ directly from the virtual document for the profession. We then perform a maximum likelihood estimate for the probabilities for a person’s professions. Similar as in pLSI, we use expectation maximization to iteratively approximate these likelihoods.

4.3.4 Experimental Results

To match our benchmark, we map the output of all runs to scores in the range between 0 and 7. Table 6 compares various approaches on two measures: Accuracy-2 (fraction of scores that differ from the human judgments by at most 2) and the Average Score Difference (ASD) from the judgments. In our publication (Bast, Buchhold, and Haussmann, 2015), we also examine rank-based measures and a second predicate, *nationality*, and we perform a refined analysis where we examine which popularity brackets are the hardest.

In this document, we also restrict ourselves to the most important approaches: *First* is a simple baseline that works as follows: The profession (resp. nationality) that is first mentioned literally in the Wikipedia article of the person gets the highest score 7, all others get a score of 0. *LLDA* (Ramage et al., 2009) is the approach from the literature which is most similar to ours (see Section 4.3.2). We compare these approaches against the three model discussed in Section 4.3.3: the *Binary Classifier*, *Weighted Indicators* and our *Generative Model*. We also consider a combination of the generative model and the binary classifier (*Combined*) and a *Control Group* that consists of another batch of human crowd-sourcing workers which we asked to judge the triples from the Benchmark again.

	Accuracy-2	ASD
First	53%	2.71
LLDA	68%	1.86
Binary Classifier	61%	2.09
Weighted Indicators	75%	1.61
Generative Model	77%	1.61
Combined (GM + Classifier)	80%	1.52
Control Group (Humans)	94%	0.92

Table 6: **Accuracy-2 and Average Score Difference (ASD) for the *profession* predicate.** Adapted from (Bast, Buchhold, and Haussmann, 2015) and the slides of the talk at SIGIR 2015 held by the author of this document.

The results reported in Table 6 are mostly as expected. Remarkably, weighted indicators and our generative model both significantly outperform LLDA. This is certainly due to the fact that our models were, unlike LLDA, developed for this exact use case, but it also shows how less complex models can be very effective when properly tailored towards the task at hand. The baseline, (*First*), is relatively weak. While this is partially due to its simplicity, there is another major factor at work: Just like our *Binary Classifier*, it produces a binary decision and we map that to 0 or 7. This is obviously not ideal if we compare against gradual scores. Especially, mapping scores to 2 or 5 would always be beneficial for the Accuracy-2 measure. This is highlighted by the fact that the *Binary*

Classifier, while weak on its own, can be used in combination with the other models to further improve the score.

As one would expect, a control group of human judges also did not perfectly agree with the scores obtained from the initial judges. This is not surprising given the way scores were derived, and given the nature of the task, where minor differences in scores lie in the eye of the beholder. However, their performance shows that there is a natural limit to how well approaches are expected to perform and that this cap is below 100% accuracy or 0 ASD.

Still, even our best approaches cannot fully compete with humans. The results obtained by participants in our triple scoring task at the 2017 WSDM Cup confirm this. The best of the 21 teams that submitted valid results reach Accuracy-2 of 82%, 80%, 80% and ASD of 1.50, 1.59, 1.61. These results are very similar to our own strongest models, whereas only the best of them (Ding, Wang, and Wang, 2017) slightly outperforms them. That winning system built an ensemble consisting of the three models introduced by us and described in this document (binary classifier, weighted indicator words and the generative model) and a fourth component based on paths in the Freebase KB. The key idea behind this fourth component is to build another binary classifier, but this time with paths that connect two entities as features and not based on their occurrences in text. While this approach slightly outperforms our models in isolation (and our basic combination), the fact that the competition winner still relies on our models in its ensemble further emphasizes their value.

Given how difficult it was for us, and all participants in the WSDM cup, to reach the same quality as human judges, we suspect that another step towards significantly better scores is very hard. In the extreme case, an approach would require nearly the same level of sophistication in understanding the text corpus that humans reach – and that is still a dream of the future for NLP in general.

Still, our scores are already good enough to yield a huge improvement in the Broccoli search engine. The way we integrated them so far is very simplistic: We only use the relevance scores for queries without a text part, because queries with text part already have useful (albeit imperfect) scores. In absence of scores for a text match, we bring our relevance scores for matching triples and the popularity scores from FreebaseEasy (see Section 4.1.3) for the entity to the same range and compute the sum of the two. The development of an advanced integration, in particular one that also makes use of the relevance scores for queries with text (where they are just as valid), is interesting future work.

4.4 Survey: Semantic Search on Text and Knowledge Bases

Semantic Search is a very broad field. Even when we set aside search in images, audio, or video and restrict ourselves to search on text and knowledge bases, there is an abundance of research. On first sight, some lines of work are so different that they appear to be entirely unrelated to one another. We have published an extensive survey (Bast, Buchhold, and Haussmann, 2016) in which we shed light onto that situation. Our main contribution is a classification of systems along two dimensions: the kind of data to search in, and the kind of queries to answer.

	Keyword Search	Structured Search	Natural Lang. Search
Text	Keyword Search on Text	Structured Data Extraction from Text	Question Answering on Text
Knowledge Bases	Keyword Search on Knowledge Bases	Structured Search on Knowledge Bases	Question Answering on Knowledge Bases
Combined Data	Keyword Search on Combined Data	Semi-Struct. Search on Combined Data	Question Answering on Combined Data

Figure 12: **Classification of systems for semantic search on text and knowledge bases according to (Bast, Buchhold, and Haussmann, 2016).**

Figure 12 visualizes our classification of systems. Three kinds of data and three kinds of queries create nine combination and thus classes to which we assign all systems. Within each class, we can now identify basic techniques that are used and refined across all systems. In the light of our classification, similarities and differences between the various lines of research finally appear intuitive and reasonable.

At first glance, the dimensions of the classification are pretty self explanatory. However, there is sometimes only a fine line between keyword search and natural language search (question answering). For example, a query like *building europe destroyed fire* can be interpreted to simply retrieve documents in which all those keywords occur. However, it could also be seen as an abbreviation of the natural language question *What are buildings in Europe that were destroyed in a fire?* This becomes even harder to decide for queries

like *buildings in europe destroyed in fire*. Thus, two systems may interpret the same query in very different ways.

For our classification, we distinguish between keyword and natural language search, not on the exact formulation of the query, but based on its intent and how systems interpret the queries. If they just try to intelligently match keywords (or synonyms and other semantically related words) somewhere in the data, we class them as doing keyword search. If they infer a narrative as it would be expressed in a question, we class them as doing natural language search. This distinction ensures that we classify systems that use similar techniques together with each other.

Another important contribution of the survey is that it allows to quickly find relevant datasets and benchmarks for a particular kind of data or search. For each kind of data, we list the most important datasets and provide statistics that allow assessing their differences on first sight. For each of our nine classes, we list the most important benchmarks and highlight the strongest contestants. This provides valuable insight concerning the strength of approaches within a class but also helps to distinguish the various benchmarks and competitions evolving around semantic search - something that can be as confusing as the plethora of systems to newcomers to the field of semantic search.

The survey also provides another perspective on the work presented in this document by widening the focus and regarding it alongside different approaches to semantic search. Our work included in this thesis is classified as *Semi-Structured Search on Combined Data* and so is most related work discussed here. However, some of that related work also falls into the other classes that deal with combined data, especially into *Keyword Search on Combined Data*. Classic SPARQL engines, naturally, fall into the class *Structured Search on Knowledge Bases*.

In the future, we think the work presented in this document may also find application in question answering. Currently, there are not many systems for *Question Answering on Combined Data* and the class mostly contains work like IBM's Watson (Ferrucci et al., 2013), where data is not truly combined but instead many subsystems work on different kinds of data. Our work, however, may be more interesting for extending today's approaches from the class *Question Answering on Knowledge Bases* so that they can also make use of a text corpus linked to their KB. State-of-the-art systems, like Aquu (Bast and Haussmann, 2015), typically generate many candidate queries based on typical query patterns. Then they use learning-to-rank techniques to choose the best query. Therefore, features are based on the generated query candidate and the original question to answer. In principle, it is thinkable that query patterns can be extended to include text search, e.g., in the form of SPARQL+Text queries, and our work would come in very handy for that.

The survey is intended to also serve as a tutorial for newcomers to the field. Therefore, we provide a focused overview of basic natural language processing tasks in semantic search. For each task, we present the idea, list state-of-the-art approaches and important benchmarks, and also point out where the tasks find application within semantic search.

Similarly, we also cover advanced techniques: ranking, indexing, ontology matching and merging, and inference. For these, however, concrete algorithms often only find application within few or even a single system.

5 Conclusion

We have introduced SPARQL+Text search for queries on a text corpus linked to a knowledge base. In Section 4.1 we have shown how effective the kind of search is, on the example of our search engine, Broccoli. Further, we have presented QLever, a highly efficient, open-source query engine with full SPARQL+Text support. Since this is an extension to SPARQL, the de-facto standard for knowledge-base queries, we provide a standard interface to the search capabilities of Broccoli that can easily be set up for special-purpose combinations of knowledge bases and text from various domains.

With Broccoli and especially QLever, we have developed the most efficient query engines for this kind of search and comparable paradigms. In our experiments in Section 4.2, we have shown that QLever works on the Freebase+ClueWeb dataset (23.4 billion word, 3.3 billion entity occurrences, and 3.1 billion KB triples) without problems. Further, we have produced working and publicly available software and demos that allow reproducing our experiments and integrating them as components in larger systems (e.g., for question answering).

In the near future, there are many technical aspects that can be improved. State-of-the-art SPARQL engines still contain many valuable features that have no corresponding components in QLever. Especially index updates (INSERT and UPDATE operations) and distributed setups have not been considered by us, yet. While we do not see fundamental problems to adopt practice-proven techniques from pure SPARQL engines, their integration is not trivial either. Apart from that, some advanced parts of SPARQL (e.g., the OPTIONAL and GROUP BY keywords) are not supported yet but should be straightforward to implement. Further, QLever uses a very simplistic LRU (least recently used) cache for queries and sub-queries that keeps a fixed number of queries. This simplistic approach works very well in our use cases and experiments so far, but due to the lack of large-scale non-synthetic query sets, we have not yet performed extensive experiments with millions of queries as realistically asked by users or applications. With such a dataset, we could certainly find ways to fine-tune our cache and improve performance further.

As a more fundamental development, our relevance scores for triples, as presented in Section 4.3, have to be integrated in a retrieval model. Currently, the scores improve the public demo of the Broccoli search engine, but they only resolve glaring issues on simple queries like the example query for *actors*. Thus, we are far from fully using their potential. Every query that involves a type-like predicate (like a person’s *profession*) can benefit from our machine-learned relevance scores, no matter how complex it is or if it also involves text search. Such an integration includes consideration of the scores in ranking functions, but is not limited to it: First of all, the scores also have to be represented as part of the knowledge-base data. This is not possible as part of the usual triples. Currently, Broccoli uses auxiliary data structures that use a key based on the entire triple to access its score. Alternatives, like allowing quadruples and adding

mediator objects (as often done to represent n-ary relations in KBs; see Section 4.1.3) all have their up- and downsides and many options should carefully be considered.

The most important kind of future work, however, presents the biggest challenge: a truly user-friendly way to ask queries. The UI of Broccoli is not ideal for non-expert users and the SPARQL+Text language of QLever is clearly not intended for end-users. The possibility to formulate queries as natural language questions and/or abbreviated as keywords (similarly to how Google interprets many semantic queries today, e.g., *What are buildings in Europe?* and *buildings in europe* yield the same KB result; see Section 2.4) sounds very promising. Systems for question answering on knowledge bases (see Section 4.4) already solve a strongly related problem decently well. There is active research on extensions that include a linked text corpus, but there is still a long way to perfection and the addition of text makes this already hard task even harder. There are many ways to formulate the same query and depending on those, the quality of the results our systems return may vary by a large margin. In general, however, our systems are very well suited to extend today's approaches for question answering on KBs to also include text search tomorrow.

If we can make the final step, and find a way to get from convenient user input to near-perfect queries, our work can also improve upon state-of-the-art commercial web search engines for such queries: By searching a KB and a text corpus in a combined way, we can answer queries that cannot be answered on either source alone. This still fills a gap in the current repertoire of all large search engines. However, what may seem like a tiny final step, may as well be the biggest challenge of all.

6 References

- Anh, V. N. and A. Moffat (2010). “Index Compression using 64-Bit Words.” In: *Softw., Pract. Exper.* 40.2, pp. 131–147. URL: <http://dx.doi.org/10.1002/spe.948>.
- Auer, S., C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives (2007). “DBpedia: A Nucleus for a Web of Open Data.” In: *ISWC/ASWC*, pp. 722–735. URL: http://dx.doi.org/10.1007/978-3-540-76298-0_52.
- Balog, K., P. Serdyukov, and A. P. de Vries (2010). “Overview of the TREC 2010 Entity Track.” In: *TREC*. URL: <http://trec.nist.gov/pubs/trec19/papers/ENTITY.OVERVIEW.pdf>.
- Balog, K., A. P. de Vries, P. Serdyukov, P. Thomas, and T. Westerveld (2009). “Overview of the TREC 2009 Entity Track.” In: *TREC*. URL: <http://trec.nist.gov/pubs/trec18/papers/ENT09.OVERVIEW.pdf>.
- Bast, H. and I. Weber (2006). “Type Less, Find More: Fast Autocompletion Search with a Succinct Index.” In: *SIGIR*, pp. 364–371. URL: <http://doi.acm.org/10.1145/1148170.1148234>.
- Bast, H., A. Chitea, F. M. Suchanek, and I. Weber (2007). “ESTER: Efficient Search on Text, Entities, and Relations.” In: *SIGIR*, pp. 671–678. URL: <http://doi.acm.org/10.1145/1277741.1277856>.
- Bast, H. and B. Buchhold (2013). “An Index for Efficient Semantic Full-Text Search.” In: *CIKM*, pp. 369–378. URL: <http://doi.acm.org/10.1145/2505515.2505689>.
- Bast, H. and B. Buchhold (2017). “QLever: A Query Engine for Efficient SPARQL+Text Search.” In: *CIKM*. URL: http://ad-publications.informatik.uni-freiburg.de/CIKM_qllever_BB_2017.pdf.
- Bast, H., B. Buchhold, and E. Haussmann (2015). “Relevance Scores for Triples from Type-Like Relations.” In: *SIGIR*. URL: <http://doi.acm.org/10.1145/2766462.2767734>.
- Bast, H., B. Buchhold, and E. Haussmann (2016). “Semantic Search on Text and Knowledge Bases.” In: *FnTIR* 10.2-3, pp. 119–271. URL: <http://dx.doi.org/10.1561/15000000032>.
- Bast, H., B. Buchhold, and E. Haussmann (2017). “A Quality Evaluation of Combined Search on a Knowledge Base and Text.” In: *Künstliche Intelligenz*. URL: http://ad-publications.informatik.uni-freiburg.de/KI_broccoli_quality_BBH_2017.pdf.
- Bast, H. and E. Haussmann (2013). “Open Information Extraction via Contextual Sentence Decomposition.” In: *ICSC*, pp. 154–159. URL: <http://dx.doi.org/10.1109/ICSC.2013.36>.
- Bast, H. and E. Haussmann (2015). “More Accurate Question Answering on Freebase.” In: *CIKM*, pp. 1431–1440. URL: <http://doi.acm.org/10.1145/2806416.2806472>.
- Bast, H., F. Bärle, B. Buchhold, and E. Haussmann (2012a). “A Case for Semantic Full-Text Search.” In: *JIWES at SIGIR*, p. 4. URL: <http://dl.acm.org/citation.cfm?id=2379311>.

- Bast, H., F. Baurle, B. Buchhold, and E. Haussmann (2012b). “Broccoli: Semantic Full-Text Search at your Fingertips.” In: *CoRR* abs/1207.2615. URL: <http://arxiv.org/abs/1207.2615>.
- Bast, H., F. Baurle, B. Buchhold, and E. Haußmann (2014a). “Easy Access to the Freebase Dataset.” In: *WWW*, pp. 95–98. URL: <http://doi.acm.org/10.1145/2567948.2577016>.
- Bast, H., F. Baurle, B. Buchhold, and E. Haußmann (2014b). “Semantic Full-Text Search with Broccoli.” In: *SIGIR*, pp. 1265–1266. URL: <http://doi.acm.org/10.1145/2600428.2611186>.
- Blanco, R., P. Mika, and S. Vigna (2011). “Effective and Efficient Entity Search in RDF Data.” In: *ISWC*, pp. 83–97. URL: http://dx.doi.org/10.1007/978-3-642-25073-6_6.
- Blanco, R., 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*. Vol. 2011. URL: <http://www.aifb.kit.edu/images/d/d9/EOS-SIGIR2011.pdf>.
- Blei, D. M., A. Y. Ng, and M. I. Jordan (2001). “Latent Dirichlet Allocation.” In: *NIPS*, pp. 601–608. URL: <http://papers.nips.cc/paper/2070-latent-dirichlet-allocation>.
- Boldi, P. and S. Vigna (2005). “MG4J at TREC 2005.” In: *TREC*. URL: <http://mg4j.di.unimi.it>.
- Bollacker, K. D., C. Evans, P. Paritosh, T. Sturge, and J. Taylor (2008). “Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge.” In: *SIGMOD*, pp. 1247–1250. URL: <http://doi.acm.org/10.1145/1376616.1376746>.
- Broekstra, J., A. Kampman, and F. van Harmelen (2002). “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema.” In: *ISWC*, pp. 54–68. URL: http://dx.doi.org/10.1007/3-540-48005-6_7.
- Cafarella, M., A. Halevy, D. Wang, E. Wu, and Y. Zhang (2008). “WebTables: Exploring the Power of Tables on the Web.” In: *PVLDB* 1.1, pp. 538–549. URL: <http://www.vldb.org/pvldb/1/1453916.pdf>.
- Cedeño, J. P. and K. S. Candan (2011). “R²DF Framework for Ranked Path Queries over Weighted RDF Graphs.” In: *WIMS*, p. 40. URL: <http://doi.acm.org/10.1145/1988688.1988736>.
- ClueWeb (2012). “The Lemur Projekt”. URL: <http://lemurproject.org/clueweb12>.
- Cucerzan, S. (2007). “Large-Scale Named Entity Disambiguation Based on Wikipedia Data.” In: *EMNLP-CoNLL*, pp. 708–716. URL: <http://www.aclweb.org/anthology/D07-1074>.
- Delbru, R., S. Campinas, and G. Tummarello (2012). “Searching Web Data: An Entity Retrieval and High-Performance Indexing Model.” In: *J. Web Sem.* 10, pp. 33–58. URL: <http://dx.doi.org/10.1016/j.websem.2011.04.004>.
- Ding, B., Q. Wang, and B. Wang (2017). “Leveraging Text and Knowledge Bases for Triple Scoring: An Ensemble Approach.” In: *WSDM Cup*. URL: <http://www.uni-weimar.de/medien/webis/events/wsdm-cup-17/wsdmcup17-papers-final/wsdmcup17-triple-scoring/ding17-notebook.pdf>.

- Dong, X., E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang (2014). “Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion.” In: *KDD*, pp. 601–610. URL: <http://doi.acm.org/10.1145/2623330.2623623>.
- Elbassuoni, S., M. Ramanath, and G. Weikum (2012). “RDF Xpress: A Flexible Expressive RDF Search Engine.” In: *SIGIR*, p. 1013. URL: <http://doi.acm.org/10.1145/2348283.2348438>.
- Elbassuoni, S., M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum (2009). “Language-Model-Based Ranking for Queries on RDF-Graphs.” In: *CIKM*, pp. 977–986. URL: <http://doi.acm.org/10.1145/1645953.1646078>.
- Elliott, B., E. Cheng, C. Thomas-Ogbuji, and Z. M. Özsoyoglu (2009). “A Complete Translation from SPARQL into Efficient SQL.” In: *IDEAS*, pp. 31–42. URL: <http://doi.acm.org/10.1145/1620432.1620437>.
- Ferrucci, D. A., A. Levas, S. Bagchi, D. Gondek, and E. T. Mueller (2013). “Watson: Beyond Jeopardy!” In: *Artif. Intell.* 199, pp. 93–105. URL: <http://dx.doi.org/10.1016/j.artint.2012.06.009>.
- Gabrilovich, E., M. Ringgaard, and A. Subramanya (2013). “FACC1: Freebase Annotation of ClueWeb Corpora”. URL: <http://lemurproject.org/clueweb12/FACC1>.
- Guha, R., D. Brickley, and S. MacBeth (2015). “Schema.org: Evolution of Structured Data on the Web.” In: *ACM Queue* 13.9, p. 10. URL: <http://queue.acm.org/detail.cfm?id=2857276>.
- Heindorf, S., M. Potthast, H. Bast, B. Buchhold, and E. Haussmann (2017). “WSDM Cup 2017: Vandalism Detection and Triple Scoring.” In: *WSDM*, pp. 827–828. URL: <http://dl.acm.org/citation.cfm?id=3022762>.
- Hofmann, T. (1999). “Probabilistic Latent Semantic Indexing.” In: *SIGIR*, pp. 50–57. URL: <http://doi.acm.org/10.1145/312624.312649>.
- Meusel, R., P. Petrovski, and C. Bizer (2014). “The WebDataCommons Microdata, RDFa and Microformat Dataset Series.” In: *ISWC*, pp. 277–292. URL: http://dx.doi.org/10.1007/978-3-319-11964-9_18.
- Mihalcea, R. and A. Csomai (2007). “Wikify! Linking Documents to Encyclopedic Knowledge.” In: *CIKM*, pp. 233–242. URL: <http://doi.acm.org/10.1145/1321440.1321475>.
- Moerkotte, G. 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*, pp. 930–941. URL: <http://dl.acm.org/citation.cfm?id=1164207>.
- Monahan, S., D. Carpenter, M. Gorelkin, K. Crosby, and M. Brunson (2014). “Populating a Knowledge Base with Entities and Events.” In: *TAC*. URL: <http://www.languagecomputer.com/news/28/15/TAC-KBP-2014.html>.
- Neumann, T. and G. Weikum (2008). “RDF-3X: A RISC-style Engine for RDF.” In: *PVLDB* 1.1, pp. 647–659. URL: <http://www.vldb.org/pvldb/1/1453927.pdf>.

- Neumann, T. and G. Weikum (2009). “Scalable Join Processing on Very Large RDF Graphs.” In: *SIGMOD*, pp. 627–640. URL: <http://doi.acm.org/10.1145/1559845.1559911>.
- Neumann, T. and G. Weikum (2010). “The RDF-3X Engine for Scalable Management of RDF Data.” In: *VLDB J.* 19.1, pp. 91–113. URL: <http://dx.doi.org/10.1007/s00778-009-0165-y>.
- Popov, B., A. Kiryakov, D. Ognyanoff, D. Manov, and A. Kirilov (2004). “KIM - A Semantic Platform for Information Extraction and Retrieval.” In: *Natural Language Engineering* 10.3-4, pp. 375–392. URL: <http://dx.doi.org/10.1017/S135132490400347X>.
- Pound, J., P. Mika, and H. Zaragoza (2010). “Ad-hoc Object Retrieval in the Web of Data.” In: *WWW*, pp. 771–780. URL: <http://doi.acm.org/10.1145/1772690.1772769>.
- Ramage, D., D. L. W. Hall, R. Nallapati, and C. D. Manning (2009). “Labeled LDA: A Supervised Topic Model for Credit Attribution in Multi-Labeled Corpora.” In: *EMNLP*, pp. 248–256. URL: <http://www.aclweb.org/anthology/D09-1026>.
- Singhal, A. (2012). “Introducing the Knowledge Graph: Things, not Strings”. URL: <https://googleblog.blogspot.de/2012/05/introducing-knowledge-graph-things-not.html>.
- Suchanek, F. M., G. Kasneci, and G. Weikum (2007). “YAGO: A Core of Semantic Knowledge.” In: *WWW*, pp. 697–706. URL: <http://doi.acm.org/10.1145/1242572.1242667>.
- Tablan, V., K. Bontcheva, I. Roberts, and H. Cunningham (2015). “Mimir: An Open-Source Semantic Search Framework for Interactive Information Seeking and Discovery.” In: *J. Web Sem.* 30, pp. 52–68. URL: <http://www.sciencedirect.com/science/article/pii/S1570826814001036>.
- Wang, H., Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan (2009). “Semple: A Scalable IR Approach to Search the Web of Data.” In: *J. Web Sem.* 7.3, pp. 177–188. URL: <http://dx.doi.org/10.1016/j.websem.2009.08.001>.
- Weiss, C., P. Karras, and A. Bernstein (2008). “Hexastore: Sextuple Indexing for Semantic Web Data Management.” In: *PVLDB* 1.1, pp. 1008–1019. URL: <http://www.vldb.org/pvldb/1/1453965.pdf>.
- Zaragoza, H., N. Craswell, M. J. Taylor, S. Saria, and S. E. Robertson (2004). “Microsoft Cambridge at TREC 13: Web and Hard Tracks.” In: *TREC*. URL: <http://trec.nist.gov/pubs/trec13/papers/microsoft-cambridge.web.hard.pdf>.