

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
Appendix: Publications	61

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.

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:


```

graph LR
    B[Building] -- occurs with --> D[destroy* fire]
    B -- located in --> E[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>

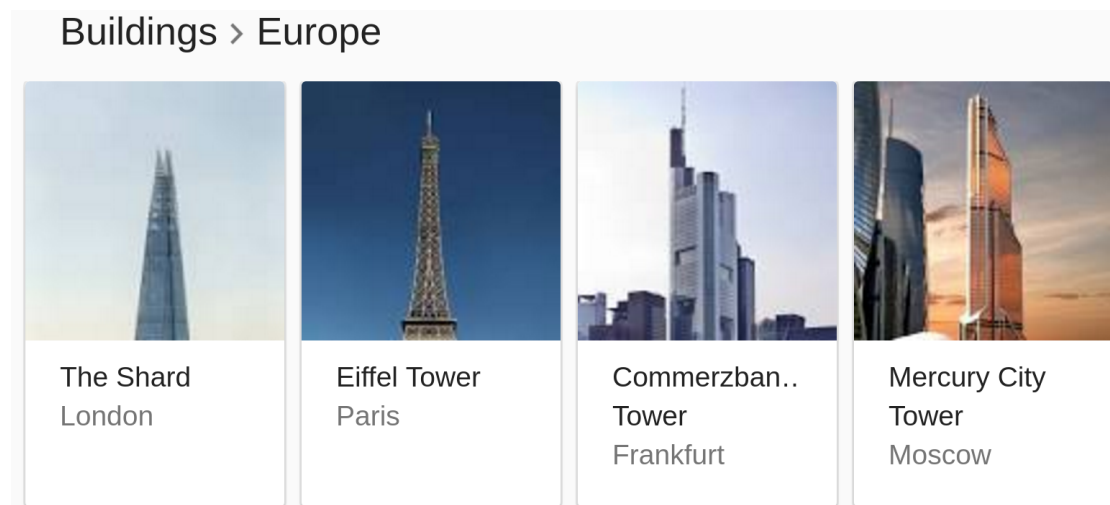


Figure 3: **Additional section at the top of the search result page for the query *buildings in Europe* from Google Search Jun 2017**

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/>

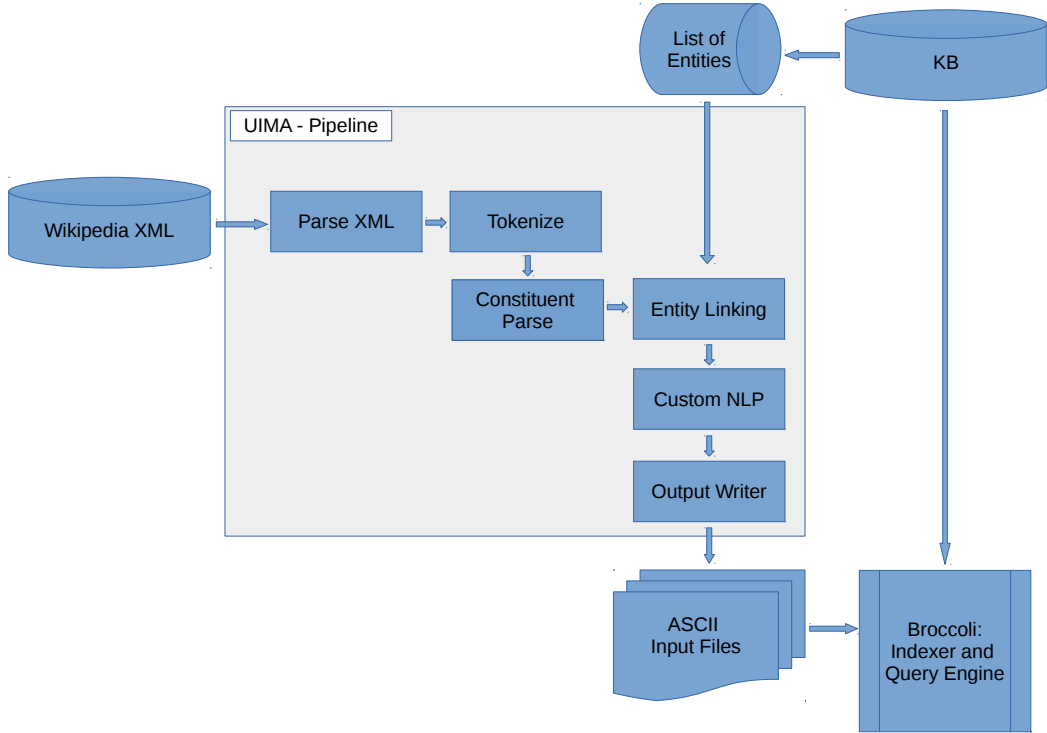


Figure 4: **The pre-processing pipeline to create an instance of the Broccoli search engine from a KB and a Wikipedia XML text corpus, slightly simplified.**

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.

Left Interface (Query: Building occurs-with tallest)		Right Interface (Query: Building occurs-with destroy* fire)	
architect, the RELATION		architect, the RELATION	
▼ Instances: Burj Khalifa (340) Empire State Building (203) Chrysler Building (104) 1 - 3 of 267		▼ Instances: Royal Opera House (58) Old Parliament Building (Quebec) (52) Pantheon (34) 1 - 3 of 167	
▼ Classes: Location (267) Structure (267) Skyscraper (226) 1 - 3 of 41		▼ Classes: Location (167) Structure (167) Tourist attraction (26) 1 - 3 of 47	
▼ Relations: located-in <Location> (252) architect <Person> (197) floors <Integer> (88) 1 - 3 of 8		▼ 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.

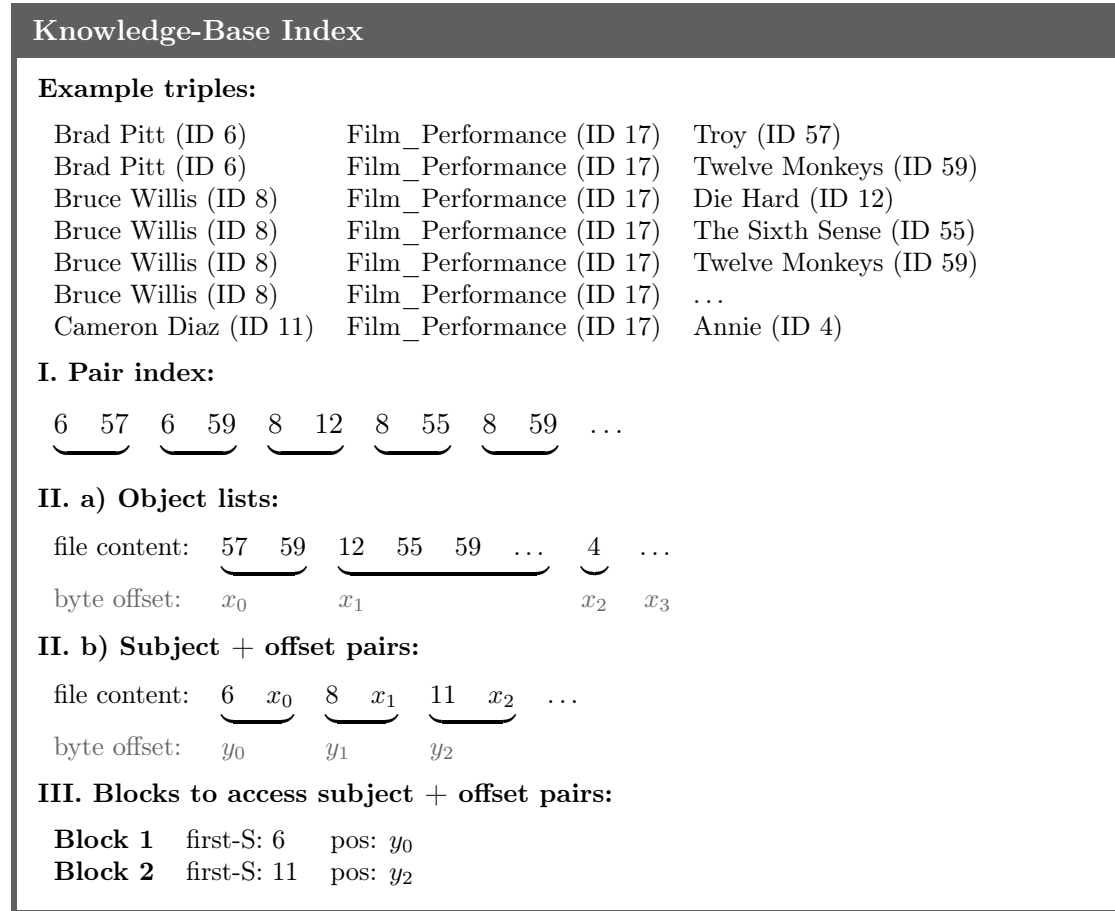


Figure 7: The components of our knowledge-base index, exemplified for the PSO permutation for a *Film Performance* predicate. Originally published in (Bast and Buchhold, 2017).

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.

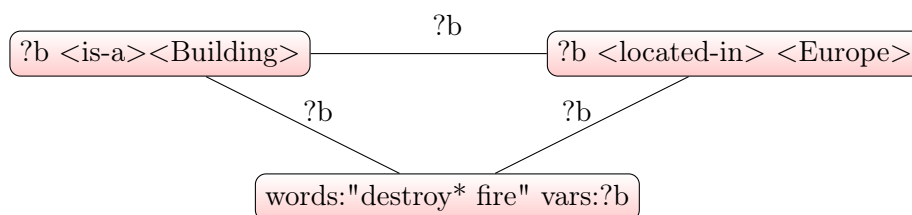


Figure 9: Graph for Query 2 from Section 1 (*buildings in Europe that were destroyed in a fire*). Cliques formed by a shared variable for a text record are collapsed into a single node.

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).

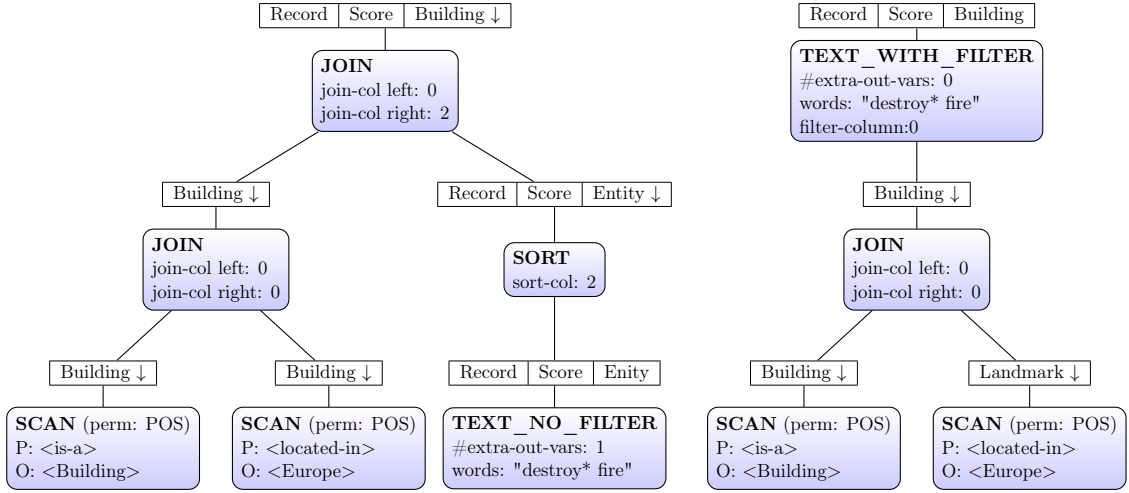


Figure 10: **Two (out of many possible) example execution trees for the query from Figure 9. The right one is smaller, because of the complex *Text with Filter* operation.**

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. Strohmman, 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. Garelkin, 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). “Sempleore: 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>.

Appendix: Publications

The following contains all publications that constitute this thesis. They are listed in chronological order and since they are embedded here as originally published, they may contain their original page numbers in addition to their page numbers within this document.

Electronic copies are available on a website:

http://ad-publications.cs.uni-freiburg.de/theses/PhD_Bjoern_Buchhold.html

A Case for Semantic Full-Text Search (JIWES at SIGIR 2012)	62
Broccoli: Semantic Full-Text Search at your Fingertips (CoRR 2012)	65
An Index for Efficient Semantic Full-Text Search (CIKM 2013)	75
Easy Access to the Freebase Dataset (WWW 2014)	85
Semantic Full-Text Search with Broccoli (SIGIR 2014)	89
Relevance Scores for Triples from Type-Like Relations (SIGIR 2015)	91
Semantic Search on Text and Knowledge Bases (FnTIR 2016)	101
WSDM Cup 2017: Vandalism Detection and Triple Scoring (WSDM 2017)	257
QLever: A Query Engine for Efficient SPARQL+Text Search (CIKM 2017)	259
A Quality Evaluation of Combined Search on a Knowledge Base and Text (KI Journal 2017)	269

A Case for Semantic Full-Text Search

(position paper)

Hannah Bast, Florian Baurle, Björn Buchhold, Elmar Haussmann

Department of Computer Science

University of Freiburg

79110 Freiburg, Germany

{bast,baeurlef,buchholb,haussmann}@informatik.uni-freiburg.de

ABSTRACT

We discuss the advantages and shortcomings of full-text search on the one hand and search in ontologies / triple stores on the other hand. We argue that both techniques have an important quality missing from the other. We advocate a deep integration of the two, and describe the associated requirements and challenges.

1. FULL-TEXT SEARCH

The basic principle of full-text search is that the user enters a (typically small) set of keywords, and the search engine returns a list of documents, in which some or all of these keywords (or variations of them like spelling variants or synonyms) occur. The results are ranked by how *prominent* these occurrences are (term frequency, occurrence in title, relative proximity, absolute importance of the document, etc.)

1.1 Document-oriented queries

This works well as long as (i) the given keywords or variants of them occur in enough of the relevant documents, and (ii) the mentioned prominence of these occurrences is highest for the most relevant documents. For example, a Google query for *broccoli* will return the Wikipedia page as the first hit, because it's a popular page containing the query word in the URL. A query for *broccoli gardening* will also work, because relevant documents will contain both of the words, most likely in a title / heading and in close proximity.

For large document collections (as in web search), the number of matching documents is usually beyond what a human can read. Then *precision* is of primary concern for such queries, not recall. The informational Wikipedia page (or a similar page) should come first, not second or fourth. And it is not important that we find *all* broccoli gardening tips on the internet.

Bottom line: *Full-text queries work well when relevant documents contain the keywords or simple variations of them in a prominent way. The primary concern is precision, not recall.*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JIWES '12 August 12 2012, Portland, OR, USA
Copyright 2012 ACM 978-1-4503-1601-9/12/08 ...\$15.00.

1.2 Entity-oriented queries

Consider the query *plants with edible leaves*. As we explain now, this kind of query is inherently problematic for full-text search engines.¹

The first problem is as follows. Relevant documents are likely to contain the words *edible leaves* or variations of them (see above). But there is no reason why they should contain the word *plants*, or variations of it like *plant* or *botany*. Rather, they will contain the name of a particular plant, for example, *broccoli*. This is exactly the kind of knowledge contained in *ontologies*, discussed in Section 2.

The second problem is that the sought-for results are not documents and also not passages in documents, but rather a list of entities, plants with a certain property. This is not only an issue of convenience for the user, but also one of *result diversity*. Even if the search engine would manage to match instances of plants (like *broccoli*) to the keyword *plant*, the problem remains that the result list contains many hits referring to one and the same (well-known) plant, while many (lesser known) plants will be missing.

Worse than that, the information for a single hit could be spread over several documents. For example, for the query *plants with edible leaves and native to Europe*, the information that a particular plant has edible leaves may be contained in one document, while the information that it is native to Europe may be contained in another document. This is beyond the capabilities of full-text search engines.

Unlike for the queries from the previous subsection, *recall* is much more important now. Precision must not be ignored, but becomes a secondary concern. For example, consider the query *apollo astronauts who walked on the moon* from the 2011 Yahoo Semsearch challenge². A user would certainly like to find all 12 entities matching this query. Regarding precision, it would be acceptable if a number of irrelevant results of the *same* order of magnitude were interspersed.

Bottom line: *Entity-oriented queries typically require ontological knowledge. High recall is of primary concern. Precision must not be ignored, but becomes secondary.*

2. ONTOLOGIES

For the purpose of this short paper, we view ontologies as collections of subject-predicate-object triples (often called *facts*), where each element of each triple has an identifier that is consistent among triples. For example, *Broccoli is*

¹Let us ignore the untypical case that a precompiled document containing those words and the result list exist.

²semsearch.yahoo.com

a Vegetable or Vegetable is-subclass-of Plant or Broccoli is-native-to Europe.

Given an ontology with sufficient information, it is easy to ask even complex queries, which require the connection of many facts, with a precisely defined semantics. For example, the query for all *plants native to europe* would require the connection of all three facts from the previous paragraph.

2.1 Obtaining the facts

The first obvious problem of ontologies is how to obtain the facts. This is easy, if the facts are already organized in a database. Then all that is needed is a conversion to the proper format. A large part of *linked open data (LOD)* [5] and hence of the BTC datasets [7] is of this kind.

Another source of ontology data are users creating machine-readable fact triples explicitly. Only a relatively small part of the BTC datasets is of this kind.

However, much (if not most) information is stored in the form of natural language text, without semantic markup. For the obvious reason that this is the primary form of communication between human beings. For example, there are thousands of documents, including several Wikipedia articles, on the web stating somewhere in a sentence that the leaves of broccoli are edible. But this information is neither contained in DBpedia, nor in current LOD, nor in BTC.

Bottom line: *Much information is available only in the form of natural language text. This is unlikely to change also in the long run. In particular, for recent and specific information.*

2.2 Information extraction

Extracting facts of the form above from natural language text is a hard problem due to the diversity and ill-definedness of natural language. This task, known as *information extraction*, is an offline process. Whatever information was failed to be extracted will not be contained in the ontology, although it should be. Whatever wrong information was extracted will be in the ontology, although it should not be.

It is exactly one of the secrets of success of full-text search that it avoids this problem in the first place. Full-text search engines simply index (almost) every word in all documents. When the document contains your keywords, you have a chance to find it. Also note how, for the sake of precision, a search engine company like Google has introduced new features (like error-tolerance or synonyms or returning) only at a point where they were virtually error-free.

In contrast, state of the art information extraction from natural language text is far from being error-free. For example, the best system on the ACE 2004 dataset for the extraction of 7 predefined relations reported a precision of 83% and a recall of 72% [11], [1]. In [2], a state of the art system that automatically identifies and extracts arbitrary relationships in a text gives an average precision of 88% and a recall of only 45%. Both systems only extract binary relationships and the extraction of multiway relationships is a significantly more complicated task [13].

Bottom line: *Fact extraction from natural language text is an offline problem with a high error rate. The typical recall is 70% or less even for popular relations.*

2.3 Consistent names

The other big problem with ontology data is consistent naming of entities and relations. LOD solves this by unifying

different names meaning the same thing via user-created links (*owl:sameAs*). This works well for popular relations and entities, but for more specific and less popular relations, such user-created links are less likely to exist.

Bottom line: *Unified names for entities and relations are feasible for a core of popular facts, but unreasonable to expect for other facts.*

3. INTEGRATION OF FULL TEXT AND ONTOLOGY DATA

In the previous two sections we have argued how a large part of the world's information is (and will be for a long time) available only as full text, while for a certain core of popular knowledge an ontology is the storage medium of choice. We therefore advocate an integration of the two types of search, which we will refer to as *semantic full-text search*.

We see *four* major research challenges associated with such a semantic full-text search. We will describe each of them in one of the following four subsections. We will also comment how we addressed them in our own prototype for an integrated such search, called *Broccoli* [3]. We encourage the reader to try our online demo available under broccoli.informatik.uni-freiburg.de

We remark that we are not claiming that our own prototype is the only way to address these research challenges.

3.1 Entity recognition in the full text

An essential ingredient of a system for semantic full-text search is the recognition of references (including anaphora) to entities from the given ontology³ in the given full text. For example, consider the following sentence: *The stalks of rhubarb are edible, but its leaves are toxic.* Both of the underlined words should map these words to the corresponding entity or entities from the given ontology, for example, dbpedia.org/resource/Rhubarb.

For reasonable query times, this kind of entity recognition has to be done *offline*. However, unlike the fact extraction described in Section 2.2, state-of-the-art methods for entity recognition achieve relatively high values for both precision and recall of around 90% [12].

Bottom line: *Offline entity recognition is an essential ingredient of semantic full-text search. The task is much simpler than full information extraction, with precision and recall values of around 90%.*

3.2 Combined Index

Typical entity-oriented queries like our *plants with edible leaves native to Europe* require three things: (1) finding entities matching the ontology part of the query (*plants native to Europe*), (2) finding text passages matching the full-text part of the query (*edible leaves*), and (3) finding occurrences of the entities from (1) that co-occur with the matches from (2).

For both (1) and (2), efficient index structures with fast query times exist. To solve (3), the solutions from (1) and (2) could be combined at query time. However, this is a major obstacle for fast query times, for two reasons. First, the entity recognition problem described in the previous section would have to be solved at query time. Second, even if the

³In particular, this could be from LOD or BTC.

final result is small, the separate result sets for (1) and (2) will often be huge, and fully materializing them is expensive.

In our own prototype *Broccoli*, we therefore propose a joint index with hybrid inverted lists that refer to both word and entity occurrences; for details, see [3].

Bottom line: *Semantic full-text search with fast query times seems to require a joint index over both the word and the entity occurrences.*

3.3 Semantic Context

For queries with a large number of hits, prominence of keyword occurrence has turned out to be a very reliable indicator of relevance. However, entity-oriented queries tend to have a long tail of hits with relatively little evidence in the document collection. Then natural language processing becomes indispensable [8].

For example, consider again the query *plants with edible leaves* and again the sentence *The stalks of rhubarb are edible, but its leaves are toxic*. This sentence is one of only few in the whole Wikipedia matching that query. But it should not count as a hit, since in it *edible* refers only to the *stalks* and not to the *leaves*. For such queries, we need an instrument for determining which words semantically “belong together”.

In our own prototype *Broccoli*, we solve this problem by splitting sentences into subsentences of words that belong together in this way. For the sentence above, after anaphora resolution this would be *The stalks of rhubarb are edible* and *Rhubarb leaves are toxic*. Again, see [3] for details.

Bottom line: *Entity-oriented queries often have hits with little evidence in the document collection. To identify those, a natural language processing is required that tells which words semantically “belong together”.*

3.4 User interface

We discuss two challenges which are particularly hard and important for semantic full-text search, especially in combination: *ease of use* and *transparency*.

A standard query language for ontology search is SPARQL [10]. The big advantage is its precise query semantics. The big disadvantage is that most users are either not willing or not able (or both) to learn / use such a complex query language. Languages like SPARQL are useful for the work behind the scenes, but not for the front-end.

On the other extreme of the spectrum is keyword search, the simplicity of which is one the secrets of success of full-text search. For full-text search, the query semantics of keyword search is reasonably transparent: the user gets documents which contain some or all of the keywords. For semantic full-text search this is no longer the case. Which part of the query was considered as an entity, which as a word, and which as a class of entities? How were these parts put in relation to each other?

The other major ingredient of transparency, besides a precise query semantics, are *results snippets*. Result snippets serve two main purposes. First, clarifying why the respective hit was returned. Second, allowing the user a quick check whether the hit is relevant.

Systems for what has become known as *ad-hoc object retrieval* [9] try to infer the query semantics from a simple keyword query. Result snippets are treated as a separate problem [6]. In existing semantic search engines on the web, they are often of low quality and little use, e.g. Falcons or

SWSE [4].

In our own prototype *Broccoli*, we use a hybrid approach. Like in keyword search, there is only a single search field. However, using it, the user can build a query, where part of the semantic structure is made explicit. This process is guided by extensive search-as-you-type query suggestions. Due to lack of space here, we refer the reader to our online demo under broccoli.informatik.uni-freiburg.de.

Bottom line: *Particular challenges for a user interface for semantic full-text search are ease of use and transparency. Of the currently existing semantic search engines, most neglect one or even both.*

4. REFERENCES

- [1] C. C. Aggarwal and C. Zhai, editors. *Mining Text Data*. Springer, 2012.
- [2] M. Banko and O. Etzioni. The tradeoffs between open and traditional relation extraction. In *ACL*, pages 28–36, 2008.
- [3] H. Bast, F. Bärle, B. Buchhold, and E. Haussmann. *Broccoli: Semantic full-text search at your fingertips*. *CoRR*, ad.informatik.uni-freiburg.de/papers, 2012.
- [4] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
- [5] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web. In *WWW*, pages 1265–1266, 2008.
- [6] R. Blanco and H. Zaragoza. Finding support sentences for entities. In *SIGIR*, pages 339–346, 2010.
- [7] Billion triple challenge dataset 2012. <http://km.aifb.kit.edu/projects/btc-2012/>.
- [8] S. T. Dumais, M. Banko, E. Brill, J. J. Lin, and A. Y. Ng. Web question answering: is more always better? In *SIGIR*, pages 291–298, 2002.
- [9] J. Pound, P. Mika, and H. Zaragoza. Ad-hoc object retrieval in the web of data. In *WWW*, pages 771–780, 2010.
- [10] E. Prud’hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [11] L. Qian, G. Zhou, F. Kong, Q. Zhu, and P. Qian. Exploiting constituent dependencies for tree kernel-based semantic relation extraction. In *COLING*, pages 697–704, 2008.
- [12] E. F. T. K. Sang and F. D. Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *CoRR*, cs.CL/0306050, 2003.
- [13] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

Broccoli: Semantic Full-Text Search at your Fingertips

Hannah Bast, Florian Baurle, Björn Buchhold, Elmar Haussmann

Department of Computer Science

University of Freiburg

79110 Freiburg, Germany

{bast,baeurle,buchholb,haussmann}@informatik.uni-freiburg.de

ABSTRACT

We present Broccoli, a fast and easy-to-use search engine for what we call semantic full-text search. Semantic full-text search combines the capabilities of standard full-text search and ontology search. The search operates on four kinds of objects: ordinary words (e.g., *edible*), classes (e.g., *plants*), instances (e.g., *Broccoli*), and relations (e.g., *occurs-with* or *native-to*). Queries are trees, where nodes are arbitrary bags of these objects, and arcs are relations. The user interface guides the user in incrementally constructing such trees by instant (search-as-you-type) suggestions of words, classes, instances, or relations that lead to good hits. Both standard full-text search and pure ontology search are included as special cases. In this paper, we describe the query language of Broccoli, the main idea behind a new kind of index that enables fast processing of queries from that language as well as fast query suggestion, the natural language processing required, and the user interface. We evaluated query times and result quality on the full version of the English Wikipedia (40 GB XML dump) combined with the YAGO ontology (26 million facts). We have implemented a fully functional prototype based on our ideas and provide a web application to reproduce our quality experiments. Both are accessible via <http://broccoli.informatik.uni-freiburg.de/repro-corr/>.

1. INTRODUCTION

In this paper, we describe a novel implementation of what we call *semantic full-text search*. Semantic full-text search combines traditional *full-text search* with structured search in knowledge databases or *ontology search* as we call it in this paper.

In traditional full-text search you type a (typically short) list of keywords and you get a list of documents containing some or all of these keywords, hopefully ranked by some notion of relevance to your query. For example, typing *broccoli leaves edible* in a web search engine will return lots of web pages with evidence that broccoli leaves are indeed edible.

In ontology search, you are given a knowledge database which you can think of as a store of subject-predicate-object triples. For example, *Broccoli is-a plant* or *Broccoli native-to Europe*. These triples can be thought of to form a graph of entities (the nodes) and relations (the edges), and ontology search allows you to search for subgraphs matching a given pattern. For example, find all plants that are native to Europe.

Many queries of a more “semantic” nature require the combination of both approaches. For example, consider the query *plants with edible leaves and native to Europe*, which will be our running example in this paper. A satisfactory answer for this query requires the combination of two kinds of information. First, a list of plants native to Europe. This is hard for full-text search but a showcase for ontology search, see above. Second, for each plant the information whether its leaves are edible or not. This kind of information can be easily found with a full-text search for each plant, see above. But it is quite unlikely (and unreasonable) to be contained in an ontology, for reasons explained in Section 2.3.

The basic principle of our combined search is to find *contextual* co-occurrences of the words from the full-text part of the query with entities matching the ontology part of the query. Consider the sentence: *The stalks of rhubarb are edible, but its leaves are toxic*. Assume for now that we can recognize entities from the ontology in the full text (we come back to this in Section 3.2). In this case, the two underlined words both refer to *rhubarb*, which our ontology knows is a plant that is native to Europe. Obviously, this sentence should *not* count as evidence that *rhubarb leaves are edible*. We handle this by decomposing each sentence into what we call its *contexts*: the parts of the sentence that “belong” together. In this case *the stalks of rhubarb are edible* and *rhubarb leaves are toxic*. An arc from the query tree now matches if and only if its elements co-occur in one and the same context.

Figures 1 and 2 show screenshots of our search engine in action for our example query. The figures and their captions also explain how the query can be constructed incrementally in an easy way and without requiring knowledge of a particular query language on the part of the user. We encourage the reader to try our online demo that is accessible via <http://broccoli.informatik.uni-freiburg.de/repro-corr/>.

Words

Classes:

Garden plant	(24)
House plant	(17)
Crop	(16)

1 - 3 of 28

Instances:

Broccoli	(58)
Cabbage	(34)
Lettuce	(23)

1 - 3 of 421

Relations:

occurs-with	<Anything>
cultivated-in	<Location> (67)
belongs-to	<Plant family> (58)

1 - 3 of 7

Your Query:

```


Plant
├── occurs-with ── edible leaves
└── native-to ─── Europe
  
```

Hits: 1 - 2 of 421

Broccoli

Ontology: Broccoli
 Broccoli: is a **plant**; native to **Europe**.

Document: Edible plant stems
 The **edible** portions of **Broccoli** are the stem tissue, the flower buds, as well as the **leaves**.



Cabbage

Ontology: Cabbage
 Cabbage: is a **plant**; native to **Europe**.

Document: Cabbage
 The only part of the **plant** that is normally **eaten** is the **leafy** head.




Figure 1: A screenshot of the final result for our example query. The box on the top right visualizes the current query as a tree. There is always one node in focus (shown in bold), in this case, the root of the tree. The large box below shows the hits grouped by instance (of the class from the root node) and ranked by relevance (if Broccoli is among the hits, we always rank it first). Evidence both from the ontology and the full text is provided. For the latter, a whole sentence is shown, with parts outside of the matching context grayed out. With the search field on the top left, the query can be extended further. The four boxes below provide context-sensitive suggestions that depend on the current focus in the query, here: suggestions for subclasses of plants, suggestions for instances of plants that lead to a hit, suggestions for relations to further refine the query. One of the suggestions is always highlighted, in this case the *cultivated-in* relation. It can be directly added to extend the query by pressing Return.

1.1 Our contribution

Broccoli supports a subset of SPARQL¹ (essentially trees with a single free variable at the root) for the ontology part of queries. Moreover, it allows a special *occurs-with* relation that can be used to specify co-occurrence of a class (e.g., *plant*) or instance (e.g., *Broccoli*) with an arbitrary combination of words, instances, and further subqueries. Both traditional full-text search and pure ontology search are subsumed as special cases. This gives a very powerful query language. See Section 4 for details.

For the *occurs-with* relation, we provide a novel kind of pre-processing that decomposes sentences into *contexts* of words that belong together. In particular, this considers enumerations and sub-clauses. Previous approaches have used co-occurrence in a whole paragraph or sentence, or based on word proximity; all of these often give poor results. See Section 3 for details.

We present the key idea behind a novel kind of index that supports fully interactive query times of around 100 milliseconds and less for a collection as large as the full English Wikipedia (40 GB XML dump, 418 million contexts of the kind just described). Previous approaches, including adaptations of the classic inverted index, yield query times on the order of seconds or even minutes for the kind of queries

we support on collections of this size. See Section 2.1 for related work, and Section 5 for details.

All the described features have been implemented into a fully functional system with a comfortable user interface. There is a single search field, as in full-text search, and suggestions are made after each keystroke. This allows the user to incrementally construct semantic full-text queries without prior knowledge of a query language. Results are ranked by relevance and grouped by instance, and displayed together with context snippets that provide full evidence for why that particular instance is shown. See Figures 1 and 2 for an example, and Section 6 for details.

We provide experimental results on the result quality for the English Wikipedia combined with the YAGO ontology [20]. For the quality results, we used 46 Queries from the SemSearch List Search Track (e.g., *Apollo astronauts who walked on the Moon*), 15 queries from the TREC 2009 Entity Track benchmarks (e.g., *Airlines that currently use Boeing 747 planes*) and 10 lists from Wikipedia (e.g. *List of participating nations at the Winter Olympic Games*). We allow reproducing our results at <http://broccoli.informatik.uni-freiburg.de/repro-corr/>. See Section 7 for the details of our experiments.

We want to remark that the natural language processing, the index, and the user interface behind Broccoli are com-

¹<http://www.w3.org/TR/rdf-sparql-query>

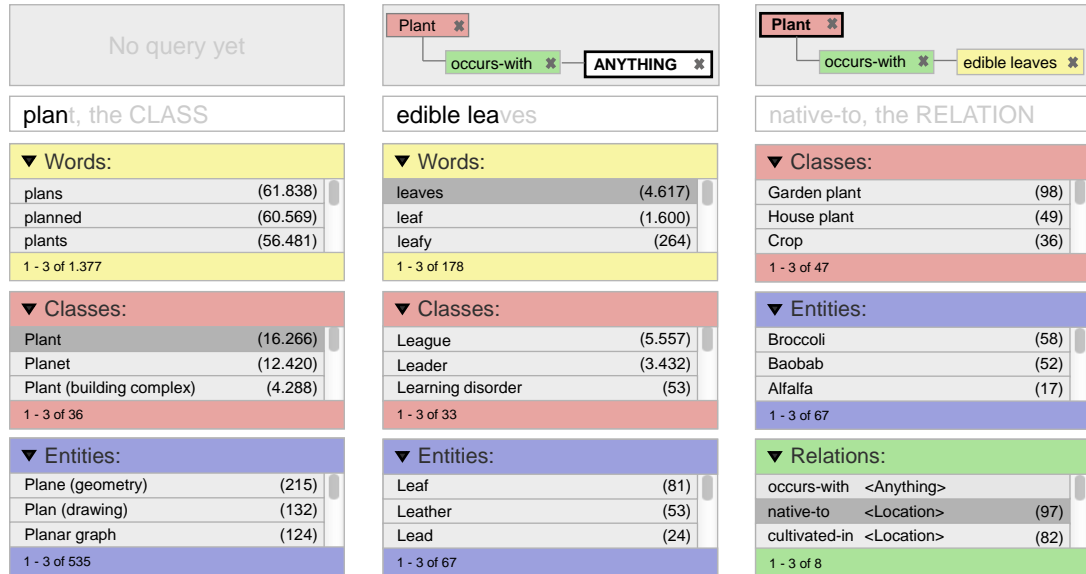


Figure 2: Snapshots of the query, search field, and suggestion boxes for three stations in the construction of our example query. Column 1: At the beginning of the query, after having typed *plan*. Column 2: After the class *plant* has been selected and the *occurs-with* relation has been added and having typed *edible lea*. Column 3: After having selected *edible leaves*. The focus automatically goes back to the root node.

plex problems each on their own. The contribution of this paper is the overall design of the system, the basic ideas for each of the mentioned components, an implementation of a fully functional prototype based on these ideas, and a first performance and quality evaluation providing a proof of concept. Optimization of the various components is the next step in this line of research; see Section 8.

2. RELATED WORK

Putting the work presented in this paper into context is hard for two reasons. First, the literature on semantic search technologies is vast. Second, “semantic” means so many different things to different researchers. We roughly divide work in this broad area into four categories, and discuss each category separately in the following four subsections.

2.1 Combined ontology and full-text search

Ester [7] was the first system to offer efficient combined full-text and ontology search on a collection as large as the English Wikipedia. Broccoli improves upon Ester in three important aspects. First, Ester works with inverted lists for classes and achieves fast query times only on relatively simple queries. Second, Ester does not consider contexts but merely syntactic proximity of words / entities. Third, Ester’s simplistic user interface was ok for queries with one relation, but practically unusable for more complex queries.

Various other systems offering combinations of full-text and ontology search have been proposed. Semplore [22] supports a query language similar to ours. However, elements from the ontology are not recognized in their contexts, but

there is simply one piece of text associated with each instance (which would correspond to a single large context in our setting). Queries are processed with a standard inverted index, and no particular UI is offered. In Hybrid Search [8], the full text and the ontology are searched separately with standard methods (Lucene and Sesame), and then the results are combined. There is no particular natural language processing. Concept Search [15] adds information about identified noun phrases and hyponyms to the index. Queries are bags of words, which are interpreted semantically. The query processing uses standard methods (Lucene), with very long inverted lists for the semantic index items. GoNToggle [14] combines full text with annotations which are searched separately and then combined, similarly as in [8]. Queries are bags of words. There is no full ontology search and no particular natural language processing. Faceted Wikipedia Search [16] offers a user interface with similarities to ours. However, the query language is restricted, there is nothing comparable to our contexts but only a small abstract per entity like in [22], and query processing is DB-based and very slow, despite the relatively small amount of data. SIREN² provides an integration of pure ontology search into Lucene. How to combine the then possible full-text and ontology searches is up to the user of the framework. Finally, systems like [23] try to interpret a given keyword query semantically and translate it into a suitable SPARQL query for pure ontology search.

²<http://siren.sindice.com>

2.2 Systems for entity retrieval

Entity retrieval is a line of research which focuses on search requests and corresponding result lists centered around entities (instead of around documents, as in traditional search). Since 2009, there is also a corresponding Entity Track at TREC³. The tasks of this track are both simpler and harder than what we aim at in this paper.

They are harder because the overall goal is entity retrieval from *web pages*. The ClueWeb09 collection introduced at TREC 2009 is 25 TB of text. The relative information content is, however, low as is typical for web contents. Moreover, identifying a representative web page for an entity is part of the problem.

To make the tasks feasible at all under these circumstances, the queries are relatively simple. For example, *Airlines that currently use Boeing 747 planes*.⁴ Even then the tasks remain very hard, and, for example, *NDCG@R* figures average only around 30% even for the best systems [4].

Broccoli queries can be trees of arbitrary degree and depth. All entities that have a Wikipedia page are supported. And, most importantly, the query process is interactive, providing the user with *instant* feedback of what is in the collection and why a particular result appears. This is key for constructing queries that give results of high quality.

The price we pay is a more extensive pre-processing assuming a certain “cleanliness” of the input collection. Our natural language processing currently requires around 1600 core hours on the 40 GB XML dump of the English Wikipedia. And Wikipedia’s rule of linking the first occurrence of an important entity in an article to the respective Wikipedia article helps us for an entity recognition of good quality; see Section 3.2. Bringing Broccoli’s functionality to web search is a very reasonable next step, but out of scope for this article.

Another popular form of entity retrieval is known as *ad-hoc object retrieval* [18]. Here, the search is on structured data, as discussed in the next subsection. Queries are given by a sequence of keywords, similar as in full-text search, for example, *doctors in barcelona*. Then query interpretation becomes a non-trivial problem; see Section 2.4.

2.3 Information extraction and ontology search

Systems for ontology search have reached a high level of sophistication. For example, RDF-3X can answer complex SPARQL queries on the Barton dataset (50 million triples) in less than a second on average [17].

As part of the Semantic Web / Linked Open Data [9] effort, more and more data is explicitly available as fact triples. The bulk of useful triple data is still harvested from text documents though. The information extraction techniques employed range from simple parsing of structured information (for example, many of the relations in YAGO or DBpedia [2] come from the Wikipedia info boxes) over pattern matching (e.g., [1]) to complex techniques involving non-trivial natural language processing like in our paper (e.g., [5]). For a relatively recent survey, see [19].

Our work differs from this line of research in two important aspects: (1) the full text remains part of the index that

is searched at query time; and (2) our system is fully interactive and keeps the human in the loop in the information extraction process. This has the following advantage:

Ontologies are good for facts like *which plants are native to which regions, who was born where on which date, etc.* Such facts are easy to define and can be extracted from existing data sources in large quantity and with reasonable quality. And once in the ontology, they are easily combinable, permitting queries that would not work with full-text search.

But for more complex facts like our *broccoli has edible leaves*, it is the other way round. They are easy to express and search in full text, but tedious to define, include, and maintain in an ontology. Let alone the problem of guessing the right relation names when searching for them.

By keeping the full text, we can leverage the intelligence of the user at query time. The query *Plant occurs-with edible leaves* does not specify the type of the relation between the occurrence of the plant and the occurrence of the words *edible* and *leaves*. Yet a moment’s thought reveals that it is quite likely that a context matching these elements gives us what we want. Similarly as in full-text search, there is often no need to be overly precise in order to get what you want. And just like the result snippets in full-text search, Broccoli’s result snippets provide instant feedback on whether the listed plant is really one with edible leaves.

Finally, if information extraction is desired nevertheless, Broccoli can be a useful tool for interactively exploring the collection with respect to the desired information, and for formulating appropriate queries.

2.4 Systems for question answering

Question answering (QA) systems provide similar functionality as our semantic full-text search. The crucial difference is that questions can be asked in natural language, which makes the answering part much harder. The system is burdened with the additional and very complex task of “translating”, in one way or the other, the given natural language query into a more formal query or queries that can be fed to a search engine and / or a knowledge database.

The perfect QA system would obviate the need for a system like ours here. But research is still far from achieving that goal. All state-of-the-art QA systems, including the big commercial ones, are specialized to quite particular kinds of questions. For example, Wolfram Alpha works perfectly for *Which cities in China have more than 10 million inhabitants*, but does not work if *more* is replaced by *less* or *China* by *Asia*, and does not even understand the question *Which plants have edible leaves*. IBM’s Watson was tuned for finding the single most probable entity when given one of the (intentionally obscured) clues from the Jeopardy! game. And both of these systems lack transparency: it is hard to predict whether a question will be understood correctly, it is hard to understand the reasons for a missing or wrong answer, and there is no possibility of interaction or query refinement.

For our semantic full-text search both the query language and the relation between a given query and its result are well-defined and maximally transparent to the user; see the discussion in Section 2.3. The price we pay is query formulation in a non-natural language. The success of full-text search has shown that as long as the language is simple enough, it can work.

³<http://ilps.science.uva.nl/trec-entity>

⁴In our framework these are queries with two nodes and one *occurs-with* edge.

3. INPUT DATA AND NATURAL LANGUAGE PRE-PROCESSING

3.1 Input data

Broccoli requires two kinds of inputs, a text collection and an ontology. The text collection consists of documents containing plain text. The ontology consists of typed *relations* with each relation containing an arbitrary set of fact triples. The subjects and objects of the triples are called *instances*. Each instance belongs to one or more *classes*. The classes are organized in a taxonomy; the root class is called *Entity*.

3.2 Entity recognition

The first step is to identify mentions of or referrals to instances from the ontology in the text documents. Consider the following sentence, which will be our running example for this section:

(S) *The usable parts of rhubarb, a plant from the Polygonaceae family, are the medicinally used roots and the edible stalks, however its leaves are toxic.*

Both *rhubarb* and *its* refer to the instance *Rhubarb* from our ontology, which in turn belongs to the classes *Plant* and *Vegetable* (among others).

Our entity recognition on the English Wikipedia is simplistic but reasonably effective. As a rule, first occurrences of entities in Wikipedia documents are linked to their Wikipedia page. When parsing a document, whenever a part or the full name of that entity is mentioned again in the same section of the document (for example, *Einstein* referring to *Albert Einstein*), we recognize it as that entity.

We resolve anaphora in an equally simplistic way. Namely, we assign each occurrence of *he*, *she*, *it*, *her*, *his*, etc. to the last recognized entity of matching gender. We also recognize the pattern *the <class>* as the entity of the document if it belongs to *<class>*, for example, *the plant* in the document of *Broccoli*.

Our results in Section 7.5 suggest that, on Wikipedia, these simple procedures give already a reasonable accuracy.

3.3 Natural language processing

The second step is to decompose document texts into what we call *contexts*, that is, sets of words that “belong” together. The contexts for our example sentence (S) from above are:

- (C1) *rhubarb, a plant from the Polygonaceae family*
- (C2) *The usable parts of rhubarb are the medicinally used roots*
- (C3) *The usable parts of rhubarb are the edible stalks*
- (C4) *however rhubarb leaves are toxic*

This will be crucial for the quality of our results, because we do not want to get *rhubarb* in our answer set when searching for *plants with edible leaves*. Note that we assume here that the entity recognition and anaphora resolution have already been done (underlined words). Also note that we do not care whether our contexts are grammatically correct and form a readable text. This distinguishes our approach from a line of research called *text simplification* [12].

In the following, we will only consider contexts that are part of a single sentence. Indeed, after anaphora resolution, it seems that most simple facts are expressed within one and the same sentence. Our evaluation in Section 7.5 confirms this assumption.

Our context decomposition consists of two parts, each described in the following subsections.

3.3.1 Sentence constituent identification (SCI)

The task of SCI is to identify the basic “building blocks” of a given sentence. For our purposes various kinds of *sub-clauses* and *enumeration items* will be important, because they usually contain separate facts that have no direct relationship to the other parts of the sentence. For example, in our sentence (S) from above, the relative clause *a plant from the Polygonaceae family* refers to *rhubarb* but has nothing to do with the rest of the sentence. Similarly, the two enumeration items *the medicinally used roots* and *the edible stalks* have nothing to do with each other (except that they both refer to *rhubarb*); in particular, *rhubarb roots* are not edible and *rhubarb stalks* are not medicinally used. Finally the part *however its leaves are toxic* needs to be considered separate from the preceding part of the sentence. As will become clear in the following, we consider these as enumeration items on the top level of the sentence.

Formally, SCI computes a tree with three kinds of nodes: *enumeration* (ENUM), *sub-clause* (SUB), and *concatenation* (CONC). The leaves contain parts of the sentence and a concatenation of the leaves from left to right yields the whole sentence again. See Figure 3 for the SCI tree of the above sentence.

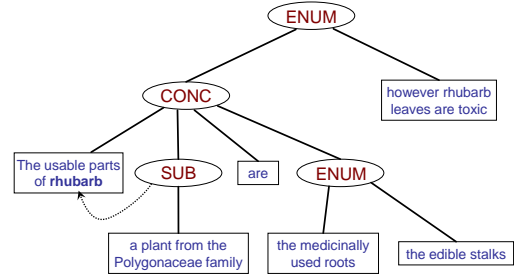


Figure 3: The SCI tree for our example sentence after anaphora resolution. The head of the sub-clause is printed in bold.

We construct our SCI trees based on the output of a state-of-the-art constituent parser. We use SENNA [13], because of its good trade-off between parse time (around 35ms per sentence) and result quality (see Section 7.5).

We transform the parse tree using a relatively small set of hand-crafted rules. Here is a selection of the most important rules; the complete list consists of only 11 rules but is omitted here for the sake of brevity. In the following description when we speak of an *NP* (noun phrase), *VP* (verb phrase), *SBAR* (subordinate clause), or *PP* (prepositional phrase) we refer to nodes in the parse tree with that tag.

(SCI 1) Mark as ENUM each node, for which the children (excluding punctuation and conjunctions) are either all *NP* or all *VP*.

(SCI 2) Mark as SUB each *SBAR*. If it starts with a word from a positive-list (e.g., *which* or *who*) define the first *NP* on the left as the *head* of this SUB; this will be used in (SCR 0) below.

(SCI 3) Mark as SUB each *PP* starting with a preposition from a positive-list (e.g., *before* or *while*), and all *PPs* at the beginning of a sentence. These SUBs have no head.

(SCI 4) Mark as CONC all remaining nodes and contract away each CONC with only text nodes in its subtree (by merging the respective text).

As our quality evaluation in Section 7.5 shows, our rules work reasonably well.

3.3.2 Sentence constituent recombination (SCR)

In SCR we recombine the constituents identified by the SCI to form our *contexts*, which will be the units for our search. Recall that the intuition is to have contexts such that only those words which “belong” together are in the same context. SCR recursively computes the following contexts from a SCI tree or subtree:

(SCR 0) Take out each subtree labeled SUB. If a head was defined for it in (SCI 2), add that head as the leftmost child (but leave it in the SCI tree, too). Then process each such subtree and the remaining part of the original SCI tree (each of which then only has ENUM and CONC nodes left) separately as follows:

(SCR 1) For a leaf, there is exactly one context: the part of the sentence stored in that leaf.

(SCR 2a) For an inner node, first recursively compute the set of contexts for each of its children.

(SCR 2b) If the node is marked ENUM, the set of contexts for this node is computed as the *union* of the sets of contexts of the children.

(SCR 2c) If the node is marked CONC, the set of contexts for this node is computed as the *cross-product* of the sets of contexts of the children.

We remark that once we have the SCI tree, SCR is straightforward, and that the time for both SCI + SCR is negligible compared to the time needed for the full-parse of the sentences.

4. QUERY LANGUAGE

Queries to Broccoli are rooted trees with arcs directed away from the root. The root is either a class or an instance. There are two types of arcs: *ontology arcs* and *occurs-with arcs*. Both have a class or instance as source node.

Ontology arcs are labeled by a relation from the ontology. The two nodes must be classes or instances matching the source and target type of the relation. The class or instance at the target node may be the root of another arbitrary tree.

For occurs-with arcs, the target node can be an arbitrary set of words, prefixes, instances or classes. The instances or classes may themselves be the root of another arbitrary query. Example queries are given in Figures 1 and 2.

To give an example of a more complex query: in Figure 1 we could replace the instance node *Europe* by a class node *Location* and add to it an *occurs-with* arc with the word *equator* in its target node. The intention of this query would be to obtain plants with edible leaves native to regions at or near the equator.

5. INDEX AND QUERY PROCESSING

The index and query processing of Broccoli are described in detail in [6]. In this section, we summarize why standard indexes are not suited for Broccoli and describe the main idea behind our new index.

There are sophisticated systems for both, full-text search and search in ontologies. Since our queries combine both tasks, three ways to answer our queries using those system come to mind: (1) incorporate ontology information into an inverted index; (2) incorporate full-text information into a triple store; (3) use an inverted index for the full-text part of the query, a triple store for the ontology part of the query, and then combine the results somehow.

Neither approach is perfectly suited for our use-case. In a nutshell, approach (1) produces document-centric results and cannot be used to answer complex queries that involve join operations. Approach (2) needs a relation (e.g. *occurs-in-context* featuring both, words and entities) of the size of our entire index to make use of the contexts produced in our contextual sentence decomposition. Efficient queries require a special purpose index over this relation, which already goes in the direction of our approach. Finally, approach (3) will get a list of contexts as a result from the full-text index and has to derive all entities that occur in those contexts. This mapping is not trivial to achieve efficiently, especially since a full mapping from contexts to entities usually does not fit in memory for large collections. Apart from that, we allow queries that demand co-occurrence with some entity from a list that can be the root of another query (e.g. a query for politicians that are friends with an astronaut who walked on the moon). This would require a second mapping in the other direction: from entities to contexts. In summary, the two problems are: Given a list of contexts *C*, produce a list *E* of entities that occur in those contexts. Given a list of contexts *C* and an entity list *E*, limit *C* to contexts that include at least one entity from *E*.

The main idea behind our new index solves these two problems. We use what we call *context lists* instead of standard inverted lists. The context list for a prefix contains one index item per occurrence of a word starting with that prefix, just like the inverted list for that prefix would. But along with that it also contains one index item for each occurrence of an arbitrary entity in the same context as one of these words. For example, consider the context *the usable parts of rhubarb are its edible stalks*, with recognized entities underlined. And let us assume that we have an inverted list for each 4-letter prefix. Then the part of the context list for *edib** pertaining to this context (which has id, say, 14) would be:

...	C14	C14	C14	...
edib*:	#edible	#Rhubarb	#Stalk	...
...	1	1	1	...
...	8	5	9	...

The numbers in the first row are context ids. The # in the second row means that not the actual entities (with capital letters) or words are stored, but rather unique ids for them. The third row contains the score for each index item. The fourth row contains the position of the word or entity in the respective context. The context lists are sorted by context id, and, for equal context ids, by word/entity id, with entities coming after the words.

Since entity postings are included in those lists, we can easily solve the two problems introduced above. Actually, our index and query processing support many additional features like excerpt generation, suggestions, prefix search, search for documents instead of entities or ranges over values. For details on those features and a detailed description of the query processing, we again refer the reader to [6].

6. USER INTERFACE

For a convincing proof of concept for our interactive semantic search, we have taken great care to implement a fully functional and intuitive user interface. In particular, there is no need for the user to formulate queries in a language like SPARQL. We claim that any user familiar with full-text search will learn how to use Broccoli in a short time, simply by typing a few queries and following the various query suggestions. The user interface is completely written in Java using the Google Web Toolkit⁵.

The introduction and screenshots (Figures 1 and 2) have already provided a foretaste of the capabilities of our user interface. Here is a list of its most important further features:

(UI 1) Search as you type: New suggestions and results with every keystroke. Very importantly, Broccoli's suggestions for words, classes, instances, and relations are context-sensitive. That is, the displayed suggestions actually lead to hits, and the more / higher-scored hits they lead to, the higher they are ranked.

(UI 2) Pre-select of most likely suggestion: Broccoli knows four kinds of objects: words, classes, instances, and relations. Depending on where you are in the query construction, you get suggestions for several of them. A new user may be overwhelmed to understand the different semantics of the different boxes. For that reason, after every keystroke Broccoli highlights the most meaningful suggestion, which can be selected by simply pressing *Return*.

(UI 3) Visual query representation: At any time, the current query is shown as a tree, with a color code for the various elements that is consistent with the suggestion boxes.

(UI 4) Change of focus / root: A click on any node in the query tree will change the focus of the query suggestions to that node. A double-click on any class or instance node will make that node the root of the tree and re-group and re-rank the results accordingly.

(UI 5) Full history support: The forward and backward buttons of the browser can be used to undo or redo single steps of the query creation process. Furthermore the current URL of the interface can always be used to store its current state or to exchange created queries with others.

(UI 6) Tutorial: Besides some pre-built example queries, the interface also provides a tutorial mode that shows how to create a search query step by step.

7. EXPERIMENTS

7.1 Input data

Our text collection is the text from all documents in the English Wikipedia, obtained via download.wikimedia.org in January 2013. Some dimensions of this collection: 40 GB

⁵<http://code.google.com/webtoolkit>

XML dump, 2.4 billion word occurrences (1.6 billion without stop-words), 285 million recognized entity occurrences and 200 million sentences which we decompose into 418 million contexts.

As ontology we use the latest version of YAGO from October 2009. We manually fixed 92 obvious mistakes in the ontology (for example, the *noble prize* was a *laureate* and hence a *person*), and added the relation *Plant native-in Location* for demonstration purposes. Altogether our variant of YAGO contains 2.6 million entities, 19,124 classes, 60 relations, and 26.6 million facts.

7.2 Pre-processing

We use a UIMA⁶ pipeline to pre-process the Wikipedia XML. The pipeline includes self-written components to parse the Wikipedia markup, tokenize text, parse sentences using SENNA [13], perform entity-recognition and anaphora resolution (see section 3.2), and decompose the sentences (see section 3.3). We want to note that all these components can easily be exchanged. In principle, this allows Broccoli to work with any given text collection and ontology.

The full parse with SENNA was scaled out asynchronously on a cluster of 8 PCs, each equipped with an AMD FX-8150 8-core processor and 16 GB of main memory. A final non-UIMA component writes the binary index which is kept in three separate files. The file for the context lists has a size of 37 GB. The file for the relation lists has a size of 0.5 GB. And the file for the document excerpts has a size of 276 GB, which could easily be reduced to 85 GB by eliminating the redundant and debug information the file currently contains.

7.3 Computing environment

The code for the index building and query processing is written entirely in C++. The code for the query evaluation is written in Perl, Java, C++ and JavaScript. Our pre-processing components are written in C++ or Java. All performance tests were run on a single core of a Dell PowerEdge server with 2 Intel Xeon 2.6 GHz processors, 96 GB of main memory, and 6x900 GB SAS hard disks configured as Raid-5.

7.4 Query times

For detailed experiments on query times, we refer to the paper describing the index behind Broccoli [6]. In said paper, we have evaluated our system on 8,000 queries of different complexity and 35,000 suggestions. Therefore we here omit a detailed breakdown and limit ourselves to the figures reported in Table 1.

Query set	average	median	90%ile	99%ile
Hit queries	52ms	23ms	139ms	393ms
Suggestion	19ms	6ms	44ms	193ms

Table 1: Statistics of query times over 8,000 queries and 35,000 suggestions.

On our collection, 90% of the queries finish within 140ms, 99% within 400ms. Suggestions are even faster. The breakdown in [6] shows that for a combination of Wikipedia and YAGO, only queries that include text take significant time. Purely ontological queries finish within 2ms on average.

⁶<http://uima.apache.org/>

		#FP	#FN	Precision	Recall	F1	P@10	R-Prec	MAP	nDCG
SemSearch	sections	44,117	92	0.06	0.78	0.09	0.32	0.42	0.44	0.45
	sentences	1,361	119	0.29	0.75	0.35	0.32	0.50	0.49	0.50
	contexts	676	139	0.39	0.67	0.43[†]	0.25	0.52	0.45	0.48
Wikipedia lists	sections	28,812	354	0.13	0.84	0.21	0.46	0.38	0.33	0.41
	sentences	1,758	266	0.49	0.79	0.58	0.82	0.65	0.59	0.68
	contexts	931	392	0.61	0.73	0.64*	0.84	0.70	0.57	0.69
TREC	sections	6,890	19	0.05	0.82	0.08	0.28	0.29	0.29	0.33
	sentences	392	38	0.39	0.65	0.37	0.58	0.62	0.46	0.52
	contexts	297	36	0.45	0.67	0.46*	0.58	0.62	0.46	0.55

Table 2: Sum of false-positives and false-negatives and averages for other measures over all SemSearch, Wikipedia list and TREC queries for Broccoli when running on sections, sentences or contexts. For contexts, the results for the SemSearch and Wikipedia list benchmarks can be reproduced using our web application at <http://broccoli.informatik.uni-freiburg.de/repro-corr/>. *, † denotes a p-value < 0.02, < 0.003 for the two-tailed t-test against the sentences baseline.

7.5 Result quality

We performed an extensive quality evaluation using topics and relevance judgments from several standard benchmarking tasks for entity retrieval: the Yahoo SemSearch 2011 List Search Track [21], the TREC 2009 Entity Track [4] and, similarly as in [7], a random selection of ten Wikipedia featured *List of ...* pages. To allow reproducibility we provide queries and relevance judgments as well as the possibility to evaluate (and modify) the queries against a live running system for the SemSearch List Track and the Wikipedia lists at <http://broccoli.informatik.uni-freiburg.de/repro-corr/>. The TREC Entity Track queries were used for an in-depth quality evaluation that does not allow for an easy reproduction. Therefore we do not provide them in our reproducibility web application. In the following we first describe each of the tasks in more detail.

The SemSearch 2011 List Search Track consisted of 50 topics asking for lists of entities in natural language, e.g. *Apollo astronauts who walked on the Moon*. The publicly available results were created by pooling the results of participating systems and are partly incomplete. Furthermore, the task used a subset of the Billion Triple Challenge Linked Data as collection, and some of the results referenced the same entity several times, e.g. once in DBpedia and once in OpenCyc. Therefore, we manually created a new ground truth consisting of Wikipedia entities. This is possible because most topics were inspired by Wikipedia lists and can be answered completely by manual investigation. Three of the topics did not contain any result entities in Wikipedia, and we ignored one additional topic because it was too controversial to answer with certainty (*books of the Jewish canon*). This leaves us with 46 topics and a total of 384 corresponding entities in our ground truth⁷. The original relevance judgments only had 42 topics with primary results and 454 corresponding entities, including many duplicates.

The TREC 2009 Entity Track worked with the ClueWeb09 collection and consisted of 20 topics also asking for lists of entities in natural language, e.g. *Airlines that currently use Boeing 747 planes*, but in addition provided the source entity (*Boeing 747*) and the type of the target entity (*organization*). We removed all relevance judgments for pages that were not contained in the English Wikipedia; this approach

was taken before in [11] as well. This leaves us with 15 topics and a total of 140 corresponding relevance judgments.

As third benchmark we took a random selection of ten of Wikipedia’s over 2,400 manually compiled featured en.wikipedia.org/wiki/List_of_... pages⁸, e.g. the *List of participating nations at the Winter Olympic Games*. Wikipedia lists are manually compiled by humans, but actually they are answers to semantic queries, and therefore perfectly suited for a system like ours. In addition, the featured Wikipedia lists undergo a review process in the community, based on, besides other attributes, comprehensiveness. For our ground truth, we automatically extracted the list of entities from the Wikipedia list pages. This leaves us with 10 topics and a total of 2,367 corresponding entities in our ground truth⁷.

For all of these tasks we manually generated queries in our query language corresponding to the semantics of the topics. We relied on using the interactive query suggestions of our user interface, but did not fine-tune our queries towards the results. An automatic translation from natural language to our query language is part of future work (see section 8). We want to stress that our goal is not a direct comparison to systems that participated in the tasks above. For that, input, collection and relevance judgments would have to be perfectly identical. Instead, we want to show that our system allows to construct intuitive queries that provide high quality results for these tasks.

We first evaluated the impact of our context decomposition from Section 3.3 (*contexts*) on result quality, by comparing it against two simple baselines: taking each sentence as one context (*sentences*) and taking each section as one context (*sections*). Table 2 shows that compared to sentences, our contexts decrease the (large) number of false-positives significantly for all benchmarks. For the TREC benchmark even the number of false-negatives decreases. This is the case because our document parser pre-processes Wikipedia lists by appending each list item to the preceding sentence (before the SCI+SCR phase). These are the only types of contexts that cross sentence boundaries and a rare exception. For the Wikipedia list benchmark we verified that this technique did not cause any results that are in the lists from which we created the ground truth. Since the sentence level

⁷ available at <http://broccoli.informatik.uni-freiburg.de/repro-corr/>

⁸http://en.wikipedia.org/wiki/Wikipedia:Featured_lists

does not represent a true superset of our contexts we also evaluated on the section level. We can observe a decrease in the number of false-negatives (a lot of them due to random co-occurrence of query words in a section) which does not outweigh the drastic increase of the number of false-positives. Overall, context decomposition results in a significantly increased precision and F-Measure, which confirms the positive impact on the user experience that we have observed.

Considering the ranking related measures in Table 2 we see a varying influence for the context based approach. The number of cases where ranking quality improves, remains unchanged or decreases is roughly balanced. This looks surprising, especially since the increase in F-measure is significant, but the reason is simple. So far our system uses simplistic ranks, determined by mere term frequency. We plan to improve on that in the future; see Section 8. We want to stress the following though. Most semantic queries, including all from the TREC and SemSearch benchmark, have a small set of relevant results. We believe that for such queries the quality of the result set as a whole is more important than the ranking within the result set. Still, for the TREC benchmark, R-precision on contexts is 0.62 and, for the SemSearch benchmark, mean average precision is 0.45. The best run from the TREC 2009 Entity Track when restricted to the English Wikipedia had an R-precision of 0.55 as reported in [11, Table 10]. The best result for the SemSearch List Search Track was a mean average precision of 0.279 [3]. Again, these results cannot be compared directly, but they do provide an indication of the quality and potential of our system.

7.6 Error analysis

To identify areas where our system can be improved we manually investigated the reasons for the false-positives and false-negatives when using contexts. We used the TREC benchmark for this, because it has a reasonable number of queries and relevance judgments that still allow a costly manual inspection of the results. We defined the following error categories. For false-positives: (FP1) a true hit which was *missing* from the ground truth; (FP2) the words in the context have a *different meaning* than what was intended by the query; (FP3) due to an error in the *ontology*; (FP4) a mistake in the *entity recognition*; (FP5) a mistake by the *parser*. (FP6) a mistake in our *context decomposition*. For false-negatives: (FN1) there seems to be *no evidence* for this entity in the Wikipedia based on the query we used. It is possible that the fact is present but expressed differently, e.g., by the use of synonyms of our query words; (FN2) the query elements are *spread* over two or more sentences; (FN3) a mistake in the *ontology*; (FN4) a mistake in the *entity recognition*; (FN5) a mistake by the *parser*; (FN6) a mistake in our *context decomposition*.

#FP	FP1	FP2	FP3	FP4	FP5	FP6
297	55%	11%	5%	12%	16%	1%

#FN	FN1	FN2	FN3	FN4	FN5	FN6
36	22%	6%	26%	21%	16%	8%

Table 3: Breakdown of errors by category.

Table 3 provides the percentage of errors in each of these categories. The high number in FP1 is great news for us: many entities are missing from the ground truth but were found by Broccoli. Errors in FN1 occur when full-text search with our queries on whole Wikipedia documents does not yield hits, independent from our contexts. Tuning queries or adding support for synonyms can decrease this number. FP2 and FN2 comprise the most severe errors. They contain false-positives that still match all query parts in the same context but have a different meaning and false-negatives that are lost because contexts are confined to sentence boundaries. Fortunately, both numbers are quite small.

The errors in categories FP and FN 3-5 depend on implementation details and third-party components. The high number in FN3 is due to errors in our current ontology, YAGO. A closer inspection revealed that, although the facts in YAGO are reasonably accurate, it is vastly incomplete in many areas (e.g., the *acted-in* relation contains only one actor for most movies). Preliminary experiments suggest that switching to Freebase [10] in the future will solve this and improve the results considerably (see section 8). To mitigate the errors caused by entity recognition and anaphora resolution (FP4+FN4), a more sophisticated state-of-the-art approach is easily integrated. Parse errors are harder. Assuming a perfect constituent parse for every single sentence, especially those with flawed grammar, is not realistic. Still, those errors do not expose limits of our approach. We hope to enable SCI+SCR without a full-parse in the future (see Section 8). The low number of errors due to our context decomposition (FP6+FN6) demonstrates that our current approach (Section 3.3) is already pretty good. Fine-tuning the way we decompose sentences might decrease this number even further.

Naturally, an evaluation should not treat entities missing in the ground-truth in the same way as actual errors. Table 4 provides quality measures for our benchmark based on sentences and contexts under three conditions: (*original*) evaluation based on the original TREC ground-truth; (*+missing*) with the entities from FP1 added to the ground truth; (*+correct*) with the errors leading to FP and FN 3,4,5 corrected.

		F1	P@10	R-Prec	MAP
Sentences	original	0.37	0.58	0.62	0.46
	+missing	0.55	0.77	0.76	0.60
Contexts	original	0.46	0.58	0.62	0.46
	+missing	0.65	0.79	0.77	0.62
	+correct	0.86	0.94	0.92	0.85

Table 4: Quality measures on TREC 2009 queries for three different levels of corrections.

The numbers for *+correct* show the high potential of our system and motivate further work correcting the respective errors. As argued in the discussion after Table 3, many corrections are easily applied, while some of them remain hard to correct perfectly.

8. CONCLUSIONS AND FUTURE WORK

We have presented Broccoli, a search engine for the interactive exploration of combined text and ontology data. We have described the index, the natural language processing,

and the user interface behind Broccoli. And we have provided reproducible evidence that Broccoli is indeed fast and gives search results of good quality.

So far, we have implemented all the basic ideas we deemed necessary to provide a convincing proof of concept. Based on this work, there are a lot of interesting directions for future research.

The underlying ontology plays a major role for our system. By switching from YAGO to Freebase we expect a great improvement of the overall quality through a better coverage of relations and thus proposals and results (see Tables 3 and 4 in the previous section). Our current approaches to entity recognition and anaphora resolution work well, but it might be possible to further improve result quality by incorporating more elaborate state-of-the-art approaches. This would also allow the system to be more easily applied to other collections than Wikipedia (our current heuristics rely on its structure, see Section 3.2). Integrating simple inference heuristics could help to reduce the number of errors that are caused by facts that are spread over several sentences. A high-quality sentence decomposition *without* the need for an expensive and error-prone full parse should further increase result quality. While query times are already low, optimized query processing and clever caching strategies have the potential to further improve speed. To investigate how to best approach performance and quality improvements, an evaluation of Broccoli on a larger, web-like collection should provide valuable insights. Automatically transforming natural language queries into our query language could help users that are accustomed to keyword queries in constructing their queries. Finally, a user study of our UI and the whole system is an important next step.

Acknowledgments

This work is partially supported by the DFG priority program Algorithm Engineering (SPP 1307) and by the German National Library of Medicine (ZB MED).

9. REFERENCES

- [1] E. Agichtein and L. Gravano. *Snowball*: extracting relations from large plain-text collections. In *ACM DL*, pages 85–94, 2000.
- [2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC*, pages 722–735, 2007.
- [3] K. Balog, M. Ciglan, R. Neumayer, W. Wei, and K. Nørvg. Ntnu at semsearch 2011. In *Proc. of the 4th Intl. Semantic Search Workshop*, 2011.
- [4] K. Balog, A. P. de Vries, P. Serdyukov, P. Thomas, and T. Westerveld. Overview of the TREC 2009 Entity Track. In *TREC*, 2009.
- [5] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *IJCAI*, pages 2670–2676, 2007.
- [6] H. Bast and B. Buchhold. An index for efficient semantic full-text search. In *CIKM*, 2013.
- [7] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.
- [8] R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi, and D. Petrelli. Hybrid search: Effectively combining keywords and semantic searches. In *ESWC*, pages 554–568, 2008.
- [9] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
- [10] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD Conference*, pages 1247–1250, 2008.
- [11] M. Bron, K. Balog, and M. de Rijke. Ranking related entities: components and analyses. In *CIKM*, pages 1079–1088, 2010.
- [12] R. Chandrasekar, C. Doran, and B. Srinivas. Motivations and methods for text simplification. In *COLING*, pages 1041–1044, 1996.
- [13] R. Collobert. Deep learning for efficient discriminative parsing. *Journal of Machine Learning Research - Proceedings Track*, 15:224–232, 2011.
- [14] G. Giannopoulos, N. Bikakis, T. Dalamagas, and T. K. Sellis. Gontogle: A tool for semantic annotation and search. In *ESWC*, pages 376–380, 2010.
- [15] F. Giunchiglia, U. Kharkevich, and I. Zaihrayeu. Concept search. In *ESWC*, pages 429–444, 2009.
- [16] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürge, H. Düwiger, and U. Scheel. Faceted wikipedia search. In *BIS*, pages 1–11, 2010.
- [17] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
- [18] J. Pound, P. Mika, and H. Zaragoza. Ad-hoc object retrieval in the web of data. In *WWW*, pages 771–780, 2010.
- [19] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.
- [20] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *J. Web Sem.*, 6(3):203–217, 2008.
- [21] T. Tran, P. Mika, H. Wang, and M. Grobelnik. Semsearch’11: the 4th semantic search workshop. In *WWW (Companion Volume)*, 2011.
- [22] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan. Semplore: A scalable IR approach to search the web of data. *J. Web Sem.*, 7(3):177–188, 2009.
- [23] G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From keywords to semantic queries - incremental query construction on the semantic web. *J. Web Sem.*, 7(3):166–176, 2009.

An Index for Efficient Semantic Full-Text Search

Hannah Bast, Björn Buchhold
Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
{bast, buchhold}@informatik.uni-freiburg.de

ABSTRACT

In this paper we present a novel index data structure tailored towards semantic full-text search. Semantic full-text search, as we call it, deeply integrates keyword-based full-text search with structured search in ontologies. Queries are SPARQL-like, with additional relations for specifying word-entity co-occurrences. In order to build such queries the user needs to be guided. We believe that incremental query construction with context-sensitive suggestions in every step serves that purpose well. Our index has to answer queries and provide such suggestions in real time. We achieve this through a novel kind of posting lists and query processing, avoiding very long (intermediate) result lists and expensive (non-local) operations on these lists. In an evaluation of 8000 queries on the full English Wikipedia (40 GB XML dump) and the YAGO ontology (26.6 million facts), we achieve average query and suggestion times of around 150ms.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Context Analysis and Indexing—*Indexing methods*

Keywords

Semantic full-text search; Indexing; Query processing

1. INTRODUCTION

Classic full-text search is very strong for document retrieval. The query *armstrong moon* typed into a web search engine will retrieve the most relevant documents about Neil Armstrong on the Moon. The query *astronauts who walked on the moon* will retrieve documents that match those keywords, hopefully leading the user to some kind of hand-compiled list of astronauts. However, the user that formulated that query probably was not looking for a list of documents but for a list of astronauts. Improving search by going beyond purely syntactic interpretation of queries is a prevalent idea. There are many different ways to add semantics to search and no approach has yet proven itself as the only way to go.

Figure 1 shows a screenshot of our system that realizes what we call semantic full-text search. The architecture of the system behind Figure 1 is described in [4].

The query for astronauts that walked on the moon and are born no later than 1930 is answered using a combination of the YAGO [15] ontology (a structured collection of facts about entities), and the English Wikipedia (unstructured full-text, in which we identify references to entities from the ontology). The information that Neil Armstrong and Buzz Aldrin are astronauts and their dates of birth are contained in the ontology. The information which astronauts have been on the moon is not contained in the ontology but expressed in the text of various Wikipedia articles.

Ontologies usually consist of a set of fact triples and are typically searched using SPARQL [14] queries. Semantic full-text search integrates SPARQL-style ontology search and full-text search in a deep way. The index presented in this paper is specifically tailored towards this kind of search.

Constructing a query like the one in the screenshot is not trivial. Although the graphical representation is easier to understand than plain SPARQL, a user will typically not know the correct names of entities or relations. Our system relies on incremental query construction, where the user is guided by context-sensitive suggestions in every step. Apart from that, we want to generate expressive excerpts. The screenshot provides comprehensive evidence for why each entity is returned as a hit.

Neither of those features is trivial to provide. In particular, the classic inverted index is less than ideal for semantic full-text search for several reasons. First, facts from the ontology have to be integrated into the index. It is possible to add classes of entities or a limited set of relations. However, it is hard to determine what to add and the index tends to become very big. Queries with chains of relations could not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2263-8/13/10 ...\$15.00.
<http://dx.doi.org/10.1145/2505515.2505689>.

Words

Classes:

Spaceflight Person	(8)
Traveler	(7)
Person	(6)

1 - 3 of 28

Instances:

Neil Armstrong	(58)
Buzz Aldrin	(34)
Pete Conrad	(23)

1 - 3 of 3

Relations:

occurs-with	<Anything>
is-citizen-of	<Country> (3)
Born-in	<Location> (3)

1 - 3 of 7

Your Query:

```


graph LR
    A[Astronaut] -- occurs-with --> B[walk* moon]
    A -- born-on-date --> C[<= 1930]
  
```

Hits: 1 - 2 of 3

Neil Armstrong

Ontology: Neil Armstrong
 Neil Armstrong: is an **astronaut**; born on date **August 5, 1930**.

Document: Kevin Foster (Entertainer)
 Foster commented: "Now I know how **Neil Armstrong** felt when he **walked on the moon**."



Buzz Aldrin

Ontology: Buzz Aldrin
 Buzz Aldrin: is an **astronaut**; born on date **January 20, 1930**.

Document: Upper Montclair, New Jersey
 Notable current and former residents of Upper Montclair include:
Buzz Aldrin, Astronaut, second man to **walk on the moon**."




Figure 1: A screenshot of our example query. The box on the top (right) visualizes the current query as a tree. The large box below shows the hits grouped by instances that match the query root and ranked by relevance. Comprehensive evidence for each hit is provided. For matches in the text corpus, a whole sentence is shown, with parts outside of the matching context grayed out. On the left, there are suggestions for classes, instances and relations w.r.t. the current query. Suggested classes are parent classes of astronaut, relations and instances are context sensitive w.r.t. the current query. There are no word suggestions (yet), because no input has been entered into the field on the top left (yet).

be answered at all. Relational databases or triple stores resort to join operations to answer complex queries over structured data. Inverted indexes do not support joins out of the box and adding them quickly compromises response times. Second, context-sensitive suggestions, which we found crucial for query construction above, are not supported by standard inverted indexes. Finally, returning entity lists with information from multiple documents combined for each hit, is not directly supported either. In Section 4 we examine the limitations of semantic full-text search with an inverted index in more detail.

The index presented in this paper supports all of the above-mentioned features efficiently. It is designed such that sorting of and join operations on entire posting lists are avoided altogether. On top of that, we construct posting lists not for words but for prefixes, which enables context-sensitive keyword suggestions similar as in [7].

The remainder of this paper is organized as follows: In Section 2, we provide an overview over other systems that combine full-text and ontology search. In Section 3, we define the query language of semantic full-text search and all the features we support. In Section 4, we elaborate on how state-of-the-art data structures for full-text search and ontology search can be used for semantic full-text search, and the limitation of this approach. In Section 5, we present our new index data structure in detail. In Section 6, we explain how queries are processed using this index. Section 7 provides the details of our experimental evaluation.

More details about our evaluation (including currently a demo of our semantic search) can be found under <http://ad.informatik.uni-freiburg.de/publications>.

2. RELATED WORK

Semantic full-text search is related to many lines of research. However, nearly all pieces of work that deal with semantic search solve different problems. This includes complex tasks where retrieval speed is not an issue and therefore only little relation to the content of this paper exists. In the following, we concentrate on previous work that employs non-standard index data structures for efficient query processing.

ESTER [5] supports semantic full-text queries, using the HYB index for fast prefix search and completion [7]. In a nutshell, words like *person:NeilArmstrong* in combination with prefix queries like *person:** are used. ESTER is fast for simple queries, but slow for others, in particular, when join operations on large index lists are involved as happens for *person:**. Apart from that, results are document-centric.

Ad-hoc entity retrieval [13] is an alternative approach to combine information from ontologies and full text. Passages of text are explicitly associated with entities from the ontology as part of the input. For example, DBpedia [2] contains textual abstracts and descriptions for many entities. Artificial documents are created for each entity, containing all the text associated with that entity. Information about which words in these texts are related and which are not is not modeled. In [8], the authors describe a system based

on MG4J¹, a fielded inverted index. Apart from ranking, the search process does not differ much from classic full-text search and hence systems are fast. One drawback of the ad-hoc ER systems we know of is that precise, factual information from ontologies is lost or watered down because it is mixed with unstructured, possibly vague text. Additionally, complex queries, e.g. a query for *entertainers that are friends with an astronaut who walked on the moon*, cannot be answered properly with a model that associates each entity with a bag of words.

Concept Search [10] extends keyword-based full-text search by semantics. Queries that contain names of concepts should also match more specific noun phrases in documents. For example, the keywords *big animal* should match documents with occurrences of the phrase *huge dog*. This overall goal is different from ours. The authors plan to improve full-text search and are not interested in retrieving lists of entities. Only abstract classes, like *animal*, are taken into consideration. Class instances like concrete persons or movies are not considered, and neither are relations from ontologies.

GoNTogle [9] is a system that performs hybrid search over an ontology and text. While this general description would also suit our system, the actual use case is, again, different. The authors assume an ontology with statements about their text documents. A core aspect of their work concerns automatic extension of that structured data. The search process is realized using a standard Lucene index. There is no focus on entity occurrences in the text and the problem is very different from semantic full-text search. For example, a typical query is "find documents relevant to XML".

In [17], the authors present a system that is supposed to efficiently deal with queries over structured data that contains relations to text documents. For the full-text part the authors build inverted indexes using Lucene and implement join operations with the structured data stored in a database system. In comparison to the example from Figure 1, there is no way facts about Neil Armstrong can be obtained from a document about Kevin Foster (Entertainer). Apart from that, the system is not able to distinguish documents that contain the word *walk* and the word *moon* independently from documents where those words occur within the same semantic context.

There are several extensions of SPARQL by full-text search, but none that we know of that deeply integrates the two. A common extension is the ability to specify an entity via keywords contained in its name (for example, entities matching *obama*). This is provided by several systems, e.g., 4store² or Jena LARQ³. A more elaborate extension is to allow the association of arbitrary text nodes with entities. Entities can then be restricted by matching keywords in that text. The system [17], discussed in the previous paragraph, falls in that category. Semantic full-text search demands a deeper integration. In Section 4, we discuss means to extend such approaches further in order to provide a deep integration for semantic full-text search.

3. QUERY LANGUAGE AND FEATURES

In Figure 1, the user interface presents queries as trees. Those trees are already close to the queries that are processed in the background, but syntactic sugar hides some important aspects. In fact, queries are similar to SPARQL with the following restrictions and extensions: Queries have to be trees - cycles are not allowed, variables cannot be used for predicates, and we add four special relations. In particular, we add a relation *occurs-with*, a relation *has-occurrence-of* and its reversed counterpart *occurs-in*. Those three relations establish the integration of full-text search. The first one is used to specify co-occurrence of entities with words or entities, the other two are used for queries involving documents. Two example queries (on a hypothetical collection where documents = patents) are *document has-occurrence-of class:Protein* and *protein occurs-in document:Patent123*. We also add a special relation *in-range* that is used for values.

Recall our query for *astronauts that walked on the moon that are born no later than 1930* and Figure 1 for its graphical representation in the user interface. The corresponding query consists of triples, organized in the following way:

```
$1 is-a Astronaut;
$1 occurs-with walk* moon;
$1 born-on-date $2;
$2 in-range date:00000000-date:19309999
```

Note that each variable identifies a node in the query tree as the UI presents it. One variable has to be explicitly specified as root. If a triple does not have a variable in first or third place, there will be an element from the ontology for regular relations. The special relations, which model occurrences in the full-text, have keywords and/or variables on their right-hand side and support operators for OR, NOT and prefix search.

Co-occurrence is always demanded within a *context* and not within an entire document. Those contexts are bags of words that semantically belong together. They are obtained through the natural language processing described in [4] and [6] and are slightly more restrictive than sentences. An example can be found in the excerpts in the screenshot (Figure 1) from above. For the sake of the index discussed in this paper and all ideas behind the index, it is fine to think of contexts as entire sentences. Either way, the index has to deal with a large number of small documents.

The screenshot depicts two other key requirements towards our index: context-sensitive suggestions and comprehensive excerpts. We cannot hope to answer semantic queries with perfect precision. Hence, it is important to provide full evidence for why a particular hit was returned and the index has to be able to generate them without compromising query times.

Formulating SPARQL queries requires knowledge of the ontology and the exact names used for relations and entities. This can be overcome if queries are constructed incrementally and guided by suggestions. Suggestions are context-sensitive and will, if added to the query, produce queries with non-empty results. If there are many suggestions, the user can provide a prefix to filter them. Rather than only filtering by actual prefixes, we also take synonyms (*athlete* vs *sports person*) into account. See Section 6.4 for how this is done.

¹<http://mg4j.di.unimi.it/>

²<http://www.4store.org/trac/wiki/TextIndexing>

³<http://jena.sourceforge.net/ARQ/lucene-arq.html>

4. LIMITATIONS OF USING KNOWN INDEX DATA STRUCTURES

In this section, we explore how known data structures for full-text search (inverted index) and ontology search (triple stores) can be used, combined, or modified to realize semantic full-text search, and the limitations of this approach. We look at three lines of approach: (1) incorporate ontology information into an inverted index; (2) incorporate full-text information into a triple store; (3) use an inverted index for the full-text part of the query, a triple store for the ontology part of the query, and then combine the results somehow. Our approach falls in the third category.

Approach (1) can be realized with different levels of sophistication. The easiest possibility is as simple as adding additional index items for each recognized reference to an entity in the given text. If for each such reference all classes the respective entity belongs to are added to the index, simple but frequent queries of the pattern *class occurs-with word(s)* can be answered. The drawbacks include index blowup, which may be acceptable depending on the number of classes and very long inverted lists for classes like *person*. More severely, queries involving more structured data than only classes, like our running example of *astronauts that walked on the moon and are born no later than 1930*, cannot be answered using this approach. In order to answer such queries, one can either add all relevant facts together with each recognized entity reference (index size explodes when the domain includes many relevant facts) or resort to join operations like ESTER [5], which has been described in Section 2. For queries involving big classes, like *person*, and sizable collections, join operations are too slow to provide an interactive user experience. Finally, complex queries that involve entire sub-queries, e.g. *entertainers that are friends with one of the astronauts from our example*, need to either (E) access an inverted list of all entity occurrences or (M) merge inverted lists for each of the entities in the sub-query result. We have found (E) to be far superior to (M), and have implemented this approach as one of our baselines in Section 7. Using this baseline, all semantic queries are significantly slower than with our new approach, ontological queries cannot be answered at all, and complex queries are unacceptably slow.

Approach (2) consists of adding words as entities to the ontology. Adding a triple for each *word occurs-with non-word-entity* (in the same document) is not an option: We want to distinguish an entity that somewhere occurs with the word *moon* and somewhere else with the word *walk*, from one that occurs with both of them in the same context. Hence, we have to add contexts to the ontology and a relation like *occurs-in-context*. First of all, the number of triples explodes. More importantly, queries would still take long to answer because they can reach over the entire left- and the right-hand-side of this *occurs-in-context* relation. An obvious optimization is to split this relation into *word-occurs-in-context* and *entity-occurs-in-context*, since it is usually clear from the query where entities and where mere keywords are involved. Still, the relations remain huge and *entity-occurs-in-context* has to be processed entirely. In Section 7, we compare our index against a highly-performant triple store using this approach.

For approach (3), it is important to note that semantic full-text queries cannot be easily split into two parts and

combined in the end. For less complex queries, such a combination is possible. [11] is a system based on this approach. Query times in their online demo⁴ are 10 seconds and beyond for most queries. This fortifies that any efficient combination is not trivial.

For fully supporting semantic full-text search, the combination has to happen at potentially several points during the query processing. Consider the query for entertainers that are friends with one of the astronauts from our example query and that the fact about friendship is retrieved from the text and not part of our ontology. There is no way to process the full-text and ontology part independently and afterwards combine the results. How such a query is solved in detail is part of what we describe in Section 6.2.

Now consider an arbitrary point in the query processing, where results from a full-text query and an ontology query have to be combined. The result for the full-text query is a list *C* of context (sentence) ids. The result for the ontology query is a list *E* of entity ids. Depending on the query, these have to be combined in two ways: (i) compute all entities in contexts from *C* that also contain an entity from *E*; (ii) compute the subset of those entities in *E* that occur in *C*. A map from context ids to entity ids is required. This is a huge map. It can either be represented as "un-inverted" index lists for each context or always be kept in memory if there is sufficient space. In Section 7 we examine both variants. Since neither is fully satisfactory, we propose a novel index layout that includes the necessary portion of this mapping within its inverted lists.

5. THE INDEX

The new index is a joint index over ontologies and text. Queries can ask for complex combinations of information from both, as explained in Section 3. We distinguish between two kinds of lists: lists containing text postings (for words or occurrences of entities), which we call context lists, and lists containing data from ontology relations. In the following, we describe both kinds of lists. How these lists are used to answer queries is described in Section 6.

5.1 Context lists

Our input is a list of postings. Postings are 4-tuples consisting of a word or entity, a context id, a score and a position. Our new index is based on two key ideas.

The first idea is taken from [7]: use inverted lists for prefixes instead of words. This enables fast prefix search and suggestions for words to use in queries. The second idea is the main idea behind our new index. We want to solve the problem introduced in Section 4, approach (3). Therefore, we use what we call *context lists* instead of usual inverted lists.

The context list for a prefix contains one index item per occurrence of a word starting with that prefix, just like the inverted list for that prefix would. But along with that it also contains one index item for each occurrence of an entity in the same context as one of these words. Similar to prefixes, we also store such a list for each entity in the context. This helps answering queries that demand co-occurrence with an entity and no word at all.

For example, consider the context: *Neil Armstrong walked on the Moon*, with recognized entity references underlined.

⁴<http://dbpedia.neofonie.de/browse/>

Let us assume that we have an inverted list for each 4-letter prefix. Then the part of the context list for *walk** pertaining to this context (which has id, say, 30) would be:

...	C30	C30	C30	...
...	#walk	#Moon	#Neil Armstrong	...
...	1	1	1	...
...	2	5	1	...

The numbers in the first row are context ids. The # in the second row means that not the actual entities (capitalized) or words (all lower case) are stored, but rather unique ids for them. The third row contains the score for each index item. The fourth row contains the position of the word or entity in the respective context.

The context lists are sorted by context id, and, for equal context ids, by word/entity id. We ensure that entity ids are always larger than word ids by setting the most significant bit for them. This ordering is used in operations during query processing (see Section 6).

Entries for context 30 will also occur in the other lists for, say, *moon** or *arms**. Each of them contains all entity postings (two in this example) and hence this is an index blowup by the average number of entities per context. For the English Wikipedia⁵, this leads to an overall factor of 1.88 (88% in addition), which is acceptable. Note that we benefit from the small context-documents. The smaller our documents, the lower the average number of entity occurrences and hence the lower the blowup factor. If we create a single context out of each sentence, the factor is 1.93, which is a bit higher but still acceptable.

In addition to the actual index, we also produce a mapping from context id to documents. Since each original document comprises a range of context ids, this mapping is trivial to produce and can be used in queries involving documents (see Section 6) as entities of their own.

5.2 Compression

Written to disk, each list is split into separate lists for word ids, context ids, scores and positions. Each individual list is compressed as follows: Word and score lists are frequency-encoded (i.e. the most frequent element in that particular list is represented by a 0, the second most frequent one by 1, and so on). Context lists are gap-encoded, position lists are left unchanged. In order to compress those lists, we use Simple8b from [1] which offers very fast decompression at the price of a slightly non-optimal compression ratio.

Simple8b performs well if elements inside a codeword are of roughly equal size. However, our lists have quite large gaps (because contexts or sentences are very short documents), followed by a number of zero-sized gaps for the entity postings within that lists. Therefore we have added the following optimization. Instead of encoding context ids as one list of gaps, we encode it as two: One list of non-zero gaps, and a second list of the same size that contains the number of zeros following each non-zero gap. This optimization has reduced the size (see Section 7 for details on our collection) needed for context lists from 4.4 GB to 3.2 GB. For all lists combined our index still requires 12 GB. An entropy-optimal encoding of our lists would require 7.8

⁵Entity recognition works well on that corpus and includes resolution of anaphora like *he*, *his*, etc.

GB. This is the price we pay for a very fast decompression, which is important for interactive query times.

In total, that is including list offsets and codebooks for restoring frequency encoded lists, our index file has a size of 13.5 GB.

5.3 Relations

Relations are stored in the straightforward way, with one index list per relation. For example, for the relation *born-on-date*:

...	#Neil Armstrong	#Richie Ginther	#Abbey Lincoln	...
...	#date:19300805	#date:19300805	#date:19300806	...
...	1	1	1	...

Again, the # means that ids are stored, not the actual entity names. The third row are the scores, which are all 1 in our current implementation. The list is sorted by the second row, that is, by the target entity ids of the relation and by source entity ids for equal targets. Since queries may use a relation in both directions, we also store the reverse for each relation separately (with rows 1 and 2 switched, and then again sorted by the ids from the second row). Technically, this is just another relation, for example, *born-on-date* (*reversed*).

For the big *is-a* relation, we store additional offsets that allow accessing the parts for a single right-hand side directly. After all, *is-a* is usually accessed with a single class. Instead of reading the whole relation, we can read exactly those entries that we need. This case also benefits from the fact that elements with equal target entity are sorted by source entity. Since we look at elements for only one specific target entity, we can directly read the entity list for the class we are looking for.

Values

Values (e.g., dates like in the example query, or integers or floats) are translated into a string representation such that the lexicographical ordering corresponds to the actual value order. We achieve this in the usual way, by concatenating a fixed-length mantissa and exponent. Hence, the way relations with values are ordered in our index is well suited for range queries.

Additional Features

In addition to the relations that are actually part of the used ontology, we create an artificial relation *has-relations* (between entities and relation names). It is used for suggesting relations sensitive to the query tree constructed so far (see Section 6.4).

Note that we can choose to make every document an entity, too. When documents are entities, they can occur in relations. This enables faceted search or search over ontologies that include relations to text documents. For example, queries like *patent documents of company X with occurrences of protein Y* are made possible this way. The only difference lies in query processing and the treatment of special relations *occurs-with* and *has-occurrence-of* as outlined in Section 6.

6. QUERY PROCESSING

In the following description of our query processing, we focus on the general algorithm and how our index and query

language come together. Standard list operations like intersection or sub-sequence extraction (which we call filtering below) are implemented in the straightforward way.

6.1 Caching

In the previous section, we have seen that queries are composed of triples. Internally, we represent each query as a tree as depicted in Figure 2.

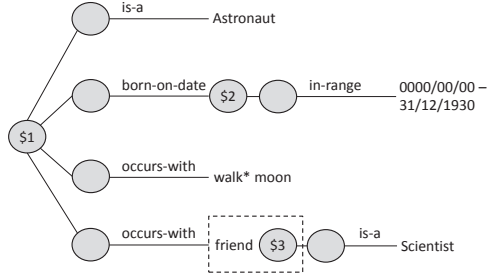


Figure 2: Query processing of an extended version of our example query. Nodes in this tree represent spots where subtree results can be computed independently and cached.

Subtrees can re-occur across queries. All intermediate results are stored in a least-recently-used (LRU) cache. The cache is implemented in the trivial way with doubly linked lists and a hash map for lookups. The LRU functionality of the cache will take care that important, recurring subtrees are kept, while uncommon subtrees are dropped.

The other cache is responsible for keeping full-text query results so that longer queries can be filtered from shorter ones. This is particularly useful when suggesting words on each keystroke (see Section 6.4). We use the same cache implementation for this cache.

6.2 Result Computation

Recall Figure 2 and that our queries are trees. In the following, we describe how each part of a query is computed when missing from the cache. If the cache is totally empty, this means we process a query tree recursively in a bottom-up fashion.

Variable nodes

(QP 1) Let E_1, \dots, E_m be the entity lists for each subtree below the current node. The result list for this node is then the intersection of the E_1, \dots, E_m , where the scores of index items with the same entity id are simply summed up.

Ontology arcs

(QP 2) For each *ontology arc*, compute the following sorted list of entities, where R denotes the relation of the arc:

(QP 2.1) For the target node t of the arc (this node can be the root of an arbitrary query again), recursively compute the result E_t , which is a sorted list of entity ids with scores.

(QP 2.2) Fetch the index list I_R for the relation R , which is sorted by target entity; see Section 5.3.

(QP 2.3) Compute the list E_R of all entities x such that $(x, y) \in R$ for some $y \in E_t$, via a straightforward intersection

of I_R with E_t . Since I_R is sorted by target entity id, this intersection can be computed efficiently in linear time. The list E_R is used for excerpt generation later (see Section 6.3) and stored along with the subtree’s result in the cache.

Although each *is-a arc* could be processed like an ordinary *ontology arc*, we can make use of their particularity. They always have a fixed class name as target. For this relation, instead of steps (QP 2.2) and (QP 2.3), directly read E_R from the index file. Remember that we stored additional offsets for that purpose as described in Section 5.3.

For each *equals arc*, lookup the entity id and use this as the only element in E_R .

Occurs-with arcs

(QP 3) For the target node of the arc, let $W = \langle w_1, \dots, w_k \rangle$ be the sequence of words, prefixes or disjunctions of words and prefixes that occur in that node without negation, and accordingly W^n for those with negation. Let $V = \{v_1, \dots, v_\ell\}$ be the set of variables in the target node. For each *occurs-with arc* then compute the following sorted list of entities.

(QP 3.1) For each v_i (which can be the root of an arbitrary query again) recursively compute its result E_i , which is a sorted list of entities.

(QP 3.2) For W compute a context list C .

(QP 3.2.1) Search the full-text cache (see below) for the longest prefix of W . If such a prefix exists, take its result C_{prefix} and extend it. If not, start with the context list for w_1 and extend this one.

(QP 3.2.2) For a fresh w_j , compute its context list Cw_j as follows: In our index, we have a context list for each k -letter prefix, for some fixed $k \geq 1$. Let I be the context list for the length- k prefix p of w_j , or, if w_j has length $< k$, for w_j .⁶ Scan over I and for each context, write all index items matching w_j (whole-word or prefix match, depending on what was specified in the query) to Cw_j , and, if at least one item matches, append all entity index items from that context, too.⁷ If w_j is a disjunction of multiple words or prefixes, compute each disjunct w_{j1}, \dots, w_{ju} accordingly. C_j is then simply the union of all Cw_{j1}, \dots, Cw_{ju} .

(QP 3.2.3) For each w_j that is a completion of an existing w_i with context list Cw_i , compute its context list Cw_j as follows: Let $I \leftarrow Cw_i$ and proceed accordingly to step 3.2.1. Hence, we avoid reading the index list from file and filter from shorter lists.

(QP 3.2.4) Extend a sequence with last element w_n and context list C_n by another element w_{n+1} with context list Cw_{n+1} in the following way: Intersect the C_n with Cw_{n+1} , such that the result list C_{n+1} contains all index items (c, e) where c is a context id that occurs in C_n and Cw_{n+1} , and e is an entity or word that occurs in context c and is an element of one of the two lists. Since the C_n and Cw_{n+1} are sorted by context ids, this can be computed in time linear to the total number of index items in the $C_n + Cw_{n+1}$.

⁶Words of length $< k$ get a context list on their own, and there are only single-word suggestions for prefixes of length $< k$.

⁷If there is no index list for a prefix of w , this means that the index items for w are contained in several index lists. In that case we could fetch all index lists $I_{p'}$ where p' is a prefix of w , and merge them. This is an expensive operation, however. Therefore we do not allow prefixes in our queries which are shorter than the prefixes from our index lists.

(QP 3.3) For each w_j^n from W^n compute a context list C_j^n . Subtract each C_j^n from C to obtain C' .

(QP 3.4) Compute a subset C'' from C' by keeping only those index items from C' with a context id such that the context contains at least one entity from each of the E_1, \dots, E_ℓ computed in Step 3.1. This can be done in time linear in $|C'| + |E_1| + \dots + |E_\ell|$, by temporarily storing each E_i in a hash map or bit-vector. The list C'' is used for excerpt generation later (see Section 6.3) and stored along with the subtree's result in the cache.

(QP 3.5) Extract all entities from C'' , aggregate the scores of all postings with the same id using summation and produce a result list that is sorted by entity id.

Has-occurrence-of arcs

(QP 4) Process each *has-occurrence-of* arc just like an *occurs-with* arc to obtain C'' (steps 3.1 through 3.4). Use the context-document mapping to obtain a list of documents from C'' . Sum up scores of all postings for contexts that belong to the same document. Since contexts are grouped by document and document ids are distributed the same way context ids are, the resulting entity list (where each entity stands for a document) is already sorted by entity id.

Occurs-in arcs

(QP 5) Process each *occurs-in* arc according to steps 3.1 - 3.3 and 3.5 of the processing of an *occurs-with* arc. Step 3.4, however, is different. Compute a subset C'' from C' by keeping only those index items from C' with a context id such that the context belongs to at least one document represented by an entity from each of the E_1, \dots, E_ℓ computed in Step 3.1. This can be computed in linear time because of the correspondence of the orderings of context ids and document ids.

In-Range arcs

(QP 6) For each *in-range* arc compute the following sorted list of entities, where R denotes the relation of the arc:

(QP 6.1) Fetch the index list I_R for the relation R , which is sorted by target entity; see Section 5.3.

(QP 6.2) At first, convert both boundaries to our representations of values of the given type. This representation has a lexicographical ordering that corresponds to the logical ordering of the values. Get the corresponding entity ids that follow the same ordering. Lookup the lower and upper bound for the range in I_R . Since I_R is sorted by target entity, we can look them up in logarithmic time using binary search. (QP 6.3) Take the left-hand-side from the range identified in I_R and sort it by entity id.

6.3 Excerpts

The query processing described above yields entity lists with scores. We have argued that evidence for entities in the result is a necessity and that we need to produce proper excerpts for them. We only need as many as shown in the UI (which can demand more excerpts when needed). Usually this means that we produce excerpts for less than 10 entities at a time.

We assume that queries are fully processed and present in the cache as described in Section 6.2. Let the result entity list be E . For each chosen entity $e \in E$ we then produce its excerpts using the following recursive algorithm.

(QPE 1) For each variable node, take the union of all excerpts produced for subtrees.

(QPE 2) For each *ontology* arc, generate the following excerpts.

(QPE 2.1) Access the list E_R that has been stored along with the subtree's result in step (QP 2.5) and select the first entry matching the current entity e . Take the target entity e' from that entry.

(QPE 2.2) Recursively generate excerpts for e' and use $E \leftarrow \{e'\}$ for that.

(QPE 2.3) Produce a textual excerpt from e , the relation name and e' . Add it to the excerpts generated recursively.

(QPE 3) For each *occurs-with* arc, generate the following excerpts.

(QPE 3.1) Access the list C'' that has been stored along with the subtree's result in step (QP 3.4). Filter it by E , using the algorithm from step 3.5 to obtain a (presumably small) context list C_{res} . This step is only done once for all entities in E .

(QPE 3.2) Filter C_{res} with e to get a context list C_e with contexts that have occurrences of this particular entity.

(QPE 3.2) Accumulate the scores⁸ for all postings with the same context id in C_e and calculate the top context. Access the original text by context id and take the according passage as excerpt. The postings in the top context of C_e include e , matching words and entities matching in subtrees further below the current arc along with their positions. Those positions decide what to highlight in the excerpts.

(QPE 3.3) If there is another subtree below the current arc, pick one of the result entities that was in the top context from the previous step as e' and recursively generate excerpts for e' using $E \leftarrow \{e'\}$.

Excerpts for *has-occurrence-of* and *occurs-in* arcs are generated accordingly. Excerpts for *ontology-arcs* featuring the *is-a* or *equals* relation can be directly produced from the query and e .

6.4 Suggestions

Query suggestions are always sensitive to the current query and the entered prefix. That prefix is a, possibly empty, part of each query (see Section 3). Since queries consist of words, instances, relations and classes, we provide separate suggestions for each of them.

In the following let $P = \langle p_1, \dots, p_n \rangle$ be the prefix for the suggestions. Note that multi word prefixes are also required to distinguish extending an *occurs-with* arc (where all words have to occur in the same context) from adding a fresh arc to the current node. Let p be the string obtained through concatenation of all elements in P without a separator. Let E be the entity list that is the solution to the current query. Note that P (and therefore p) may be empty. E may be non-existent when there is no current query, i.e., we are in the beginning of the query construction process.

Word suggestions

Word suggestions are only made when p is non-empty. The prefix has to have a minimum length of k where we have a

⁸Note that this is the only place where we currently consider the scores given to postings, and it is only for selecting the (hopefully) best excerpts.

context list for each k -letter prefix, in our index (see section 5.1). This minimum length is only relevant for word suggestions. The other suggestions discussed below, are always computed as soon as the first letter is entered.

(QPS 1) To generate word postings, compute the list W_{sugg} of word ids with scores as follows:

(QPS 1.1) Compute the context list C for the full prefix analogous to steps (QP 3.1 - QP 3.2.2) of the processing of *occurs-with arcs* using $W \leftarrow P$.

(QPS 1.2) If E exists, compute a subset C' from C by keeping only those index items from C with a context id such that the context contains at least one entity from E . The algorithm is analogous to the one depicted in step (QP 3.4). If E does not exist, use $C' \leftarrow C$.

(QPS 1.3) Lookup p_n in the vocabulary to obtain two word ids for a lower, id_{low} , and upper, id_{up} , bound on words matching the prefix.

(QPS 1.4) Compute another subset C_{words} from C' , such that $\forall (c, w) \in C_{words} : id_{low} \leq w \leq id_{up}$, where w is the word or entity id of the posting. This is done by scanning C' once, only keeping items with an id in the given range. All entities and all words that do not match the last part of the prefix are now discarded.

(QPS 1.5) Compute the list of word suggestions W_{sugg} by aggregating the elements in C_{words} by word id and, depending on settings, either sum up scores or count elements and take the count score. This is analogous to step (QP 3.5).

Finally sort W_{sugg} by score, lookup the string representation matching the ids in the vocabulary and suggest the top words.

Managing synonyms

It is possible to directly obtain entity id boundaries from p and filter only matching entities. Unfortunately, this does not allow to get Neil Armstrong with both prefixes ar^* or ne^* . Therefore we produce a mapping that contains both entries $\{\text{neilarmstrong} \rightarrow \#\text{Neil Armstrong}\}$ and $\{\text{armstrongneil} \rightarrow \#\text{Neil Armstrong}\}$ sorted by keys ($\#\text{Neil Armstrong}$ means that we, of course, store the id, not the string representation). Similarly, we can add synonyms to that mapping, e.g. $\{\text{sportsman} \rightarrow \#\text{Athlete}\}$.

We create a separate, tiny index for that mapping in the following way. Keys are sorted lexicographically and stored in a vocabulary. If there are n keys we store a vector M of n entity ids as data where $M[i]$ corresponds to the entity id that is the target of the key with id i .

(QPS 2) Compute an entity list E_{match} of entities that match p as either real prefix or as a pseudo prefix in the following way:

(QPS 2.1) Lookup p in the vocabulary to obtain a lower bound id_{low} and an upper bound id_{up} on the word ids of the keys matching the prefix.

(QPS 2.2) Create the entity list E_{match} by selecting the range between $M[id_{low}]$ and $M[id_{up}]$, and sort E_{match} by entity id.

Instance, Relation and Class suggestions

Instance suggestions are obtained from the result entity list of the query E in the obvious way. If there is a non-empty prefix p , E is intersected with E_{match} . For relation suggestions, the *has-relations (reversed)* relation is accessed with E as described in (QP 2), which yields an entity list of

relations E_R . Suggestions and prefix filtering are performed similar to instances using E_R instead of E . Context sensitive class suggestions are expensive to compute. Hence, these suggestions are the only ones that are (currently) not context-sensitive in our system. Instead we keep an extra list of all classes with scores pertaining to their number of instances and take that list as E .

7. EXPERIMENTS

We evaluated our index on three tasks: answering queries, providing excerpts, and providing suggestions. For each task we have generated multiple query sets as described below. For the answering-queries task we compare various approaches. Suggestions and excerpts cannot be provided easily or at all with some of the approaches. Details about our query sets can be found under the URL provided at the end of our introduction (Section 1).

7.1 Experimental Setup

Our text collection is the text from all documents in the English Wikipedia from January 3, 2012, obtained via download.wikimedia.org. Some dimensions of our collection: 40 GB XML dump, 2.4 billion word occurrences (1.6 billion without stop-words), 285 million recognized entity occurrences, and 334 million sentences which we decompose into 418 million contexts.

As ontology we use the latest version of YAGO from October 2009⁹. We ignore content that has no use for our application, for example, the (large) relation *during*, which provides the date of extraction for each fact. Altogether our variant of YAGO contains 2.6 million entities, 19 124 classes, 60 relations, and 26.6 million facts.

Our index is kept in three separate files. The file for the context lists has a size of 13.5 GB (see Section 5.2). The total number of postings is 1.9 times as much as in a standard full-text index. The file for the relation lists has a size of 0.5 GB. Document excerpts are simply read from a file containing the original text using precomputed byte offsets for each context.

The code for the index building and query processing is written entirely in C++. Performance tests marked with (m) were run on a single core of a Dell PowerEdge server with 2 Intel Xeon 2.6 GHz processors, 96 GB of main memory, and 6x900 GB SAS hard disks configured as Raid-5. Performance tests marked with (d) were run on a single core of a PC with a 3,6 GHz AMD processor, 4 GB of main memory and a 2 TB Seagate Barracuda 7200 hard disk. On this system, indexes do not fit in memory and hence neither in the file system cache.

Table 1 provides the average response times for eight types of queries: (Q1) full-text only, one word; (Q2) full-text only, two words; (Q3) ontology only, one arc between a class and an entity; (Q4) class occurs-with one word; (Q5) class occurs-with two words; (Q6) class ontology-arc (entity occurs-with word); (Q7) class occurs-with word and class; (Q8) class occurs-with word and (class occurs-with one word).

We synthetically generated 1,000 queries for each type. Starting from the root, we select elements as follows: For

⁹There is a more recent version, called YAGO2, but the additions from YAGO to YAGO2 are not really interesting for our search.

classes, relations and entities, pick a random ASCII prefix of length 1, and consider our system’s top 20 suggestions for the query built so far. For words, pick a random two letter prefix from the 170 most common two letter prefixes in the collection and consider the top 50 word suggestions. Pick a random one of those suggestions. If no suggestion exists, try a different random prefix. If 10 such attempts fail, start again from the root for that query. Note that using the suggestions guarantees that all queries have non-empty result sets.

Additionally we add a realistic query set (QR), which contains 46 queries that we manually constructed from the topics of the Yahoo SemSearch 2011 List Search Track [16].

7.2 Comparative evaluation

We evaluated two baselines implementing the first two approaches described in Section 4: (1) *Inv*, an inverted index that has inverted lists for each class; (2) *TS*, a triple store (we used RDF-3X [12] which is known to be very efficient) with relations *entity-occurs-in-context* and *word-occurs-in-context* added to the ontology.

Apart from our baselines, we compare two approaches that follow the query processing presented in Section 6: One is *Map*, an approach that uses normal inverted lists and additional (“uninverted”) mappings from context id to entity postings. The other is *CL*, our context lists that mix word and entity postings as presented in Section 5.

Note that the two baselines are not capable of everything our system does. The comparative evaluation only measures retrieval of result entities. Excerpt generation and suggestions cannot readily be provided by the baselines. Some query sets cannot be answered, either: Since we only included classes in the *Inv* baseline, queries that use more relations than *is-a*, are impossible for this baseline to answer. *TS* does not answer full-text only queries. Queries from *QR* require features (prefix and OR) that our baseline implementations do not provide and therefore this query set is not used in the comparative evaluation.

For all approaches marked (*m*) (for memory), we have repeated the whole experiments (including program start, no application caches involved) until no further speedup was found and hence we assume the relevant portion of the index to be in the file system’s cache. Additionally, we have evaluated *Map* and *CL* on a PC with only 4 GB of main memory and cold caches to compare setups where the index does not fit in main memory. These are marked (*d*)isk.

	Inv(m)	TS(m)	Map(m)	Map(d)	CL(m)	CL(d)
Q1	13ms	-	13ms	28ms	34ms	64ms
Q2	28ms	-	28ms	59ms	81ms	150ms
Q3	-	1ms	2ms	5ms	2ms	5ms
Q4	208ms	1.2s	78ms	45s	42ms	80ms
Q5	228ms	0.8s	37ms	2s	86ms	186ms
Q6	-	1.4s	207ms	63s	75ms	138ms
Q7	1s	2.5s	234ms	58s	58ms	115ms
Q8	2.5s	3.7s	430ms	104s	109ms	221ms
Size	16GB	87GB	11GB	11GB	14GB	14GB

Table 1: Comparison of retrieval of entity lists (avg times). No excerpts or suggestions.

Some numbers are identical. This is no coincidence since full-text only queries are entirely identical for *Inv* and *Map* and so are ontology-only queries for *Map* and *CL*. Apart from that, we observe that both baselines are not really competitive. In particular, the *Inv* baseline uses an index that contains no relational information at all but still performs poorly in comparison. Especially complex queries (Q8) are problematic, but all queries involving the long inverted lists for classes (e.g. there are 78M postings in the inverted list for *person*) have a problem.

The *Map* Approach is not too bad but relies heavily on enough RAM to hold the mappings from context to entity postings. While the *CL(d)* approach is about two times slower than *CL(m)*, *Map(d)* becomes unacceptably slow. Even if the entire index is contained in memory, *CL(m)* performs generally faster than *Map(m)* due to the better locality of access.

However, full-text only queries are obviously faster if inverted lists do not contain additional entity postings. Additionally, whenever multiple words are used in a single *occurs-with* triple, only one list has to contain all the entity postings. All others can be normal inverted lists, which accelerates reading on the one hand, and filtering for concrete words from our lists for entire prefixes on the other hand. A hybrid of *Map* and *CL* could therefore achieve slightly faster query times at the expense of additional space requirements.

7.3 Full Queries with Excerpts: Breakdown

In the previous section, we have evaluated and compared the retrieval of entity lists. In this section, we provide a breakdown by operation for the *CL* approach. Additionally, we include providing evidence hits for the top 10 entities.

	fetch(m)	fetch(d)	excerpt	agg	filter	i+m+r
Q1	20ms	48ms	2ms	5ms	7ms	1ms
Q2	57ms	124ms	<1ms	2ms	18ms	1ms
Q3	1ms	4ms	<1ms	0ms	0ms	<1ms
Q4	23ms	60ms	3ms	6ms	10ms	<1ms
Q5	59ms	152ms	<1ms	<1ms	22ms	3ms
Q6	34ms	88ms	18ms	16ms	30ms	<1ms
Q7	31ms	84ms	2ms	1ms	16ms	<1ms
Q8	61ms	165ms	8ms	13ms	32ms	1ms
QR	109ms	260ms	1ms	<1ms	59ms	5ms

Table 2: Breakdown of *CL* by operations. Total averages correspond to numbers in Table 1 plus the times from the *excerpt* column. Avg query time for QR is 182ms.

The times reported in Table 2 are for computing and showing the *hits*, that is, the contents of the large box on the right in Figure 1. Hence, they involve the steps described above in Sections 6.2 and 6.3. Queries from *QR* can be complex or involve many keywords, especially in combination with the OR operator (brother|sister|silbling*).

We observe that the bulk of the query time is spent in *fetching* lists. This includes reading the list from (*d*)isk or the file system cache (*m*), decompression, and recreating our lists of four-tuples. The other three columns provide the times for entity or document *aggregation* (in Steps QP 3.5, QP 4 and QP 5 in Section 6), *filtering* (in Steps QP

3.2.2, QP 3.4 and QPE 3.1), and intersection, merging and ranking of result lists (in Steps QP 1, QP 3.2.4 and QP 3.3). The total query time corresponds to the time listed in Table 1 plus the (negligibly small) times from the *excerpt* column.

7.4 Suggestions

Table 3 provides the times for query suggestions when formulating a query step-by-step. We differentiate here between suggestions at three different points during the query formulation process: (S1) type something in the beginning; (S2) type something after a single class has been chosen; (S3) type something after a class and a relation have been chosen and the target of the relation is active;

We take the queries from classes (Q3) and (Q4) above, create them step-by-step, and type the respective element from left to right. We always suggest words, relations, instances and classes. In the beginning (S1), no relations can be suggested. Due to the prefix length used for the context lists (see Section 5.1), word suggestions are only presented for prefixes of length 4 or greater. Note that suggestions for more complex queries (Q5-8 and QR) would be faster and not slower, because the result sets are smaller.

Active	Prefix Length				
	1	2	3	4	≥ 5
start	30ms	8ms	4ms	60ms	18ms
class	15ms	9ms	7ms	45ms	21ms
rel-target	19ms	7ms	4ms	34ms	8ms

Table 3: Query suggestion times for three different stations in the query formulation process.

The most costly operations are performed for prefixes of length 4. This is the prefix length where word suggestions are presented for the first time. Again those operations are dominated by reading the index lists from disk (45%). Word suggestions for prefix length ≥ 5 are filtered from the posting lists for smaller prefixes and are significantly faster, again. The time for presenting non-word suggestions is dominated by the time to get an entity list matching the pseudo prefix (see Step QPS 2): up to 15ms for prefix length 1, lower for longer prefixes.

8. CONCLUSIONS AND FUTURE WORK

We have presented an index that enables efficient semantic full-text search. We have argued how neither classic inverted indexes and full-text engines nor triple stores can handle our problem. For the English Wikipedia in combination with the YAGO ontology, we achieve interactive query and suggestion times of around 100ms and often less.

Text collections much larger than Wikipedia could be handled as follows. Fetching (reading and decompressing) index lists is the dominant factor. The text collections could be split into parts, and an index built for each part, on separate machines. This technique is described as collection partitioning in [3]. During query processing results can be merged when small entity lists, rather than huge posting lists, are involved.

We have experimented with a much larger ontology than YAGO (15 times more non-*is-a* facts) and have not encountered any problems. Using YAGO, lists for relations other than *is-a* contain at most 0.5 million entries. Large context lists for our Wikipedia collection usually contain about 10

million entries. On top of that, the operations performed on context lists are more complex than those performed on relation lists. Our experiments confirm that ontology queries are currently no issue for retrieval at all and that there is room for much bigger relations.

9. REFERENCES

- [1] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2):131–147, 2010.
- [2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A nucleus for a web of open data. In *ISWC*, pages 722–735, 2007.
- [3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.
- [4] H. Bast, F. Bärle, B. Buchhold, and E. Haussmann. Broccoli: Semantic full-text search at your fingertips. *CoRR*, abs/1207.2615, 2012.
- [5] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.
- [6] H. Bast and E. Haussmann. Open information extraction via contextual sentence decomposition. In *ICSC*, 2013.
- [7] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.
- [8] R. Blanco, P. Mika, and S. Vigna. Effective and efficient entity search in RDF data. In *International Semantic Web Conference (1)*, pages 83–97, 2011.
- [9] G. Giannopoulos, N. Bikakis, T. Dalamagas, and T. K. Sellis. Gontogle: A tool for semantic annotation and search. In *ESWC*, pages 376–380, 2010.
- [10] F. Giunchiglia, U. Kharkevich, and I. Zaihrayeu. Concept search. In *ESWC*, pages 429–444, 2009.
- [11] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, and U. Scheel. Faceted Wikipedia search. In *BIS*, pages 1–11, 2010.
- [12] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
- [13] J. Pound, P. Mika, and H. Zaragoza. Ad-hoc object retrieval in the web of data. In *WWW*, pages 771–780, 2010.
- [14] E. Prud’hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [15] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from Wikipedia and WordNet. *J. Web Sem.*, 6(3):203–217, 2008.
- [16] T. Tran, P. Mika, H. Wang, and M. Grobelnik. Semsearch’11: 4th Workshop on Semantic Search. In *WWW (Companion Volume)*, 2011.
- [17] H. Wang, T. Tran, C. Liu, and L. Fu. Lightweight integration of IR and DB for scalable hybrid search with integrated ranking support. *J. Web Sem.*, 9(4):490–503, 2011.

Easy Access to the Freebase Dataset

Hannah Bast, Florian Baurle, Björn Buchhold, Elmar Haußmann
Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
{bast, baeurle, buchhold, haussmann}@informatik.uni-freiburg.de

ABSTRACT

We demonstrate a system for fast and intuitive exploration of the Freebase dataset. This required solving several non-trivial problems, including: entity scores for proper ranking and name disambiguation, a unique meaningful name for every entity and every type, extraction of canonical binary relations from multi-way relations (which in Freebase are modeled via so-called mediator objects), computing the transitive hull of selected relations, and identifying and merging duplicates. Our contribution is two-fold. First, we provide for download an up-to-date version of the Freebase data, enriched and simplified as just sketched. Second, we offer a user interface for exploring and searching this data set. The data set, the user interface and a demo video are available from <http://freebase-easy.cs.uni-freiburg.de>.

Categories and Subject Descriptors

H.0 [Information Systems]: General

Keywords

Freebase; Knowledge Base; Ontology

1. INTRODUCTION

Freebase [2] is designed as an open, community-curated knowledge base. With more than 40 million topics and over 2 billion facts, it is today by far the most comprehensive publicly available source of general-knowledge facts.

The complete Freebase data is available for free use, sharing, and adaption (even commercially) under a creative commons license. The data format is N-Triples RDF, which is standard for triple data. In principle, the data can therefore be loaded into any state-of-the-art triple store and queried via standard semantic query languages such as SPARQL. Freebase also provides an own API. The query language used there is MQL.

However, when working with this raw data via SPARQL or with the Freebase API via MQL, several major usability issues arise, also for expert users. Consider the query for

winners of the Palme d'Or¹, shown in Figure 1. This appears to be a simple query, which requires only a single relation. In SPARQL one would like to write something like this:

```
select ?x where { ?x Awards-Won "Palme d'Or" }
```

But the required SPARQL query on the provided RDF data dump looks like this:

```
select ?name where {  
  ?x ns:award/award_winner/awards_won ?m .  
  ?m ns:award/award_honor/award ?a .  
  ?a ns:type/object/name "Palme d'Or"@en .  
  ?x ns:type/object/name ?name .  
}
```

Already this simple example hints at a number of usability issues. How to guess the right relation names? How to guess the right schema (the object of the *awards_won* relation is a so-called mediator object, which is linked, via another relation, to the actual award entity)? How to guess the right entity names (Palme d'Or in this case)? The results are opaque, too. Here is the link to the result for the equivalent MQL query (the complexity of which is similar to that of the SPARQL query above): <http://tinyurl.com/12pdms5>. In particular, the ranking is merely lexicographic and there are ambiguous names like *Michael Moore*. For more complex queries, e.g. <http://tinyurl.com/qzrc77j>, these problems intensify.

In contrast, the query in Figure 1 is as one would expect. As we will see later, the user interface helps in finding the proper relation names. The results are properly ranked, with the most prominent hits (directors in this case) at the top. The names of the directors are as expected, and accompanied by pictures. What is not shown is that there are 16 persons in Freebase with the name *Michael Moore*. In our version of the Freebase data set, only the (in)famous director gets exactly that name. The others are disambiguated by meaningful suffixes, e.g. *Michael Moore (Soccer Forward)*. Finally, the user interface offers suggestions for sensible ways to augment the query, e.g. by the relation *Country of nationality*.

1.1 Our contribution

We address all of the problems from the example query above, as well as several other problems that occur with typical queries and impact usability.

Entity Scores. As in standard text search, long result lists demand for a proper ranking. For example, for our example query above, we would like to have the most prominent

¹This is the highest prize at the annual Cannes Film Festival.

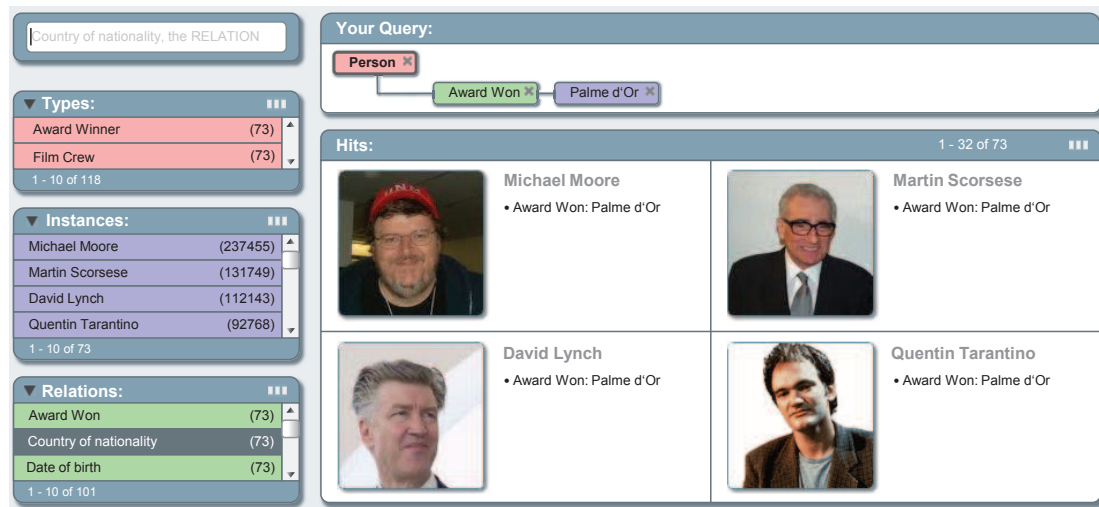


Figure 1: A screenshot of our demo system showing results for a query for winners of the Palme d'Or. For explanations of the various components and features, see the paragraph before Section 1.1 and Section 3.

people at the top. We provide a prominence score for each entity in Freebase; see Section 2.2.

Entity Names. In Freebase, each entity has a unique alpha-numerical so-called machine id or mid, e.g. */m/0jw67*. In most applications, it is desirable to also have unique names that are meaningful for humans. This is also the approach Wikipedia takes. There, entities are distinguished with suffixes. For example, *Europe* denotes the continent as expected, while the Swedish rock band with the same name is called *Europe (band)*. For the sake of consistency, these suffixes follow several rules, but ultimately they are chosen by humans. We automatically compute such names for each entity in Freebase. This is described in Section 2.3.

Mediators. In our introductory example, we have encountered the complex *awards_won* relation. It involves a mediator object that itself is related to several entities, including not only the person who won the award and the award won, but also supplementary information like the date of the award and the winning work. Still, for many queries the “main” binary relation (between the person and the award in this case) is all that is needed, and would be much easier to use. We automatically extract this binary relation from each mediator; see in Section 2.4.

Transitivity. Many relations are practically unusable when they are not closed under transitivity. The relation *Contained by* between locations is a prominent example. We compute the transitive hull for several large (manually selected) relations from Freebase; see Section 2.5.

Duplicates. Duplicate entities or types with the same or a similar name are frequent in Freebase. For example, there are four classes called *Person* or *person*. Usually, additional types with the same name have few instances and are added as a user’s mistake. The problem is aggravated by our own addition of types to the taxonomy; see the next item. We identify duplicates, merge them and give them a proper canonical name; see Section 2.6.

Taxonomy. Freebase by itself has a comparably shallow taxonomy (3,557 different types at the time of this writing) expressed via its *type/object/type* relation. However, many intuitive semantic classes like *plant* or *politician* are not types. Instead this information is available only via relations, e.g. *Profession*. We apply a set of configurable relations with objects that are to be included in the taxonomy. Our resulting taxonomy has a total of 21,042 different types. See <http://freebase-easy.cs.uni-freiburg.de> for more details.

User Interface. We provide a fully-functional user interface that allows for an interactive exploration and search using all of the features above. See Figure 1 and our description in Section 3. The demo is available under the link above; we encourage the reader to try it out.

Download. Along with the demo, we also provide our version of the Freebase data set, with all of the mentioned features, for download. A zip file (2.4 GB at the time of this writing) is available under the link above. Our data curation pipeline (see Section 2) is fully automatized. This allows us to easily update the data set on a regular basis, and thus keep pace with the continuously growing Freebase data.

We remark that some of the items above represent major research challenges. For this demo paper, we apply comparably simple and straightforward solutions. However, as can be seen from the demo, these already go a long way towards an easier access to and better usability of the Freebase data.

1.2 Related Work

There is an abundance of work on providing more convenient front ends for semantic search. See [4] for a small survey, and the many papers citing that work. None of these achieve context-sensitive query suggestions at interactive speed as in our user interface (for a data set as large as Freebase); see also [1, Section 2].

Concerning our data curation pipeline, we do not claim particular novelty for any of the components. Our contri-

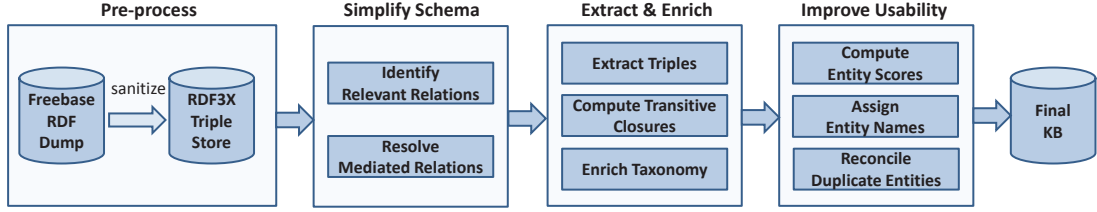


Figure 2: Architectural overview of our pipeline for a more easy-to-use version of the Freebase data set.

bution is that we have identified the major issues for the (widely used) Freebase data set, and provide a version that is much more easily accessible, and a ready-to-use demo application. We know of no comparable effort to date.

2. DESCRIPTION OF OUR PIPELINE

Freebase provides raw data dumps in the form of RDF-triples. As explained above, working with this raw data is complex for a variety of reasons. We therefore seek to simplify and enrich it in several ways. Figure 2 illustrates the general pipeline of our architecture. The various steps of the pipeline are described in the following subsections.

2.1 Data Sanitization

The raw RDF data contains redundant information as well as information which is undesired or even annoying in most use cases. We therefore first load the raw RDF data into RDF-3X [5], a fast triple store, and then extract the relations we are interested in using appropriate SPARQL queries. Namely, we omit relations with few facts (< 5) and relations that are not part of the core data in Freebase (e.g., facts in the domains *user* and *base*). Further, Freebase contains many (but not all) relations in two directions, e.g., *place-of-birth* and *people-born-here*. For all those, we only extract one direction (the one with more subjects than objects)².

2.2 Entity Scores

Scores indicating the prominence of entities are essential when ranking result entities (rank prominent entities first) and when resolving naming conflicts (assign the most prominent entity the canonical name, see Section 2.3). Intuitively, the more people talk (or write) about an entity the more prominent it is. We utilize the mentions of Freebase entities in the ClueWeb’12 Corpus³ (733M web pages) from [3] to count the number of mentions of each entity and use it as a score. Given a set of mentions M_{CW_e} of an entity e we use s_{CW} as the resulting score:

$$s_{CW}(e) = |M_{CW_e}|$$

About 4.5 million distinct entities were recognized in ClueWeb, but our knowledge base contains a total of 39.6 million distinct entities. Therefore, we additionally compute a score based on the knowledge base and its relations in the following way:

²Most applications, including our own here, can handle queries for the reverse direction without requiring a copy of it.

³<http://lemurproject.org/clueweb12/>

$$s_{KB}(e) = \sum_{r \in R} \log(\max(1, |\{x \mid (e, r, x) \in KB\}|)) + \sum_{r \in R} \log(\max(1, |\{x \mid (x, r, e) \in KB\}|))$$

R is the set of all relation types in the knowledge base and KB denotes its set of relational triples (x, r, y) . The above is the sum of the log of a per-relation out-degree and in-degree with the intuition that an entity with many incoming and outgoing relations is more prominent. The main effect of this score is as a tie-breaker, when two entities have the same number of occurrences in the ClueWeb collections or were not mentioned or recognized at all. As final score for an entity we use the sum of the ClueWeb and knowledge based score:

$$s(e) = s_{CW}(e) + s_{KB}(e)$$

2.3 Entity names

As discussed in the introduction, a unique meaningful name for each entity is highly desirable in many applications. However, the raw Freebase data only provides alpha-numerical ids and highly ambiguous names. In Wikipedia, this problem is solved manually as follows. For an ambiguous name, the most prominent entity gets the name without further additions. For example, the director from our example query in Figure 1 is called *Michael Moore*. Other contestants for the same name are distinguished by a meaningful suffix, e.g. *Michael Moore (Australian politician)*.

For the Freebase data, we automatically assign unique names as follows.⁴ If there is no name at all, use the alpha-numerical id from Freebase. Otherwise, there will be a set of candidates that compete for a name. Note that these candidates can be types (e.g. *Director*) as well as entities (e.g. *Michael Moore*). Also note that a type and an entity can have the same name in Freebase (e.g., there is a type *Person* and several entities with that name). The score for an entity is simply the score from Section 2.2. The score for a type is simply the maximum score of an entity plus the number of instances of that type. The literal name (without suffixes) then goes to the candidate with the highest score.

The remaining candidates are disambiguated as follows. If they are located in a country, they compete for the name $\langle \text{name} \rangle \langle \text{country} \rangle$. Again, the entity with the highest score gets that name. The others get an additional numerical suffix, e.g. *Berlin (United States) #2*. Entities without locational information are disambiguated using their *notable-for* relation, e.g. *Michael Moore (Soccer Forward)*. If that is not enough to achieve unique names, again a numerical suffix is

⁴Note that for most Freebase entries, there is no associated Wikipedia entry.

added. Entities that have neither locational nor *notable-for* information are disambiguated using their Freebase ids, e.g. *Maria* (*m/0760g8*).

2.4 Mediators

As explained in the introduction, Freebase realizes multi-way relations using so-called mediator objects. For example, for a fact from Freebase’s *Awards won* relation, the object is such a mediator object of type *award_honor*. This object is then related to the actual award, but also to supplementary information such as the winning work or the date of the award.

For each mediator type m (e.g. *award_honor*), we do the following. Intuitively, there are two types of mediators, which require a different approach. Namely, m either mediates between two entities in different roles (e.g. a musician and a group) or in the same role (e.g. siblings). We found the following strategy to differentiate very well between these two cases.

Consider the k relations that have m as subject.⁵ Let $n_1 \geq \dots \geq n_k$ be the number of facts in each of these k relations, sorted in decreasing order. Let r be the relation pertaining to n_1 . If $k \geq 2$ and $n_2 \geq n_1/2$, let r' be the relation pertaining to n_2 , otherwise let $r' = r$. Intuitively, r and r' are hence the most “frequent” relations, with $r = r'$ for a relation like “sibling”. It remains to “merge” r and r' to the desired binary relation and give it a proper name.

Let n be the name of the reverse direction of the relation r according to Freebase. If no such name can be obtained, try the reverse relation of r' . In the very rare event that this fails too, fall back to $n = r$. We then extract a binary relation r_m in the following way.

$$r_m = \{(s, n, o) \mid (x, r, s) \in KB \wedge (x, r', o) \in KB \wedge s \neq o\}$$

This gives us exactly one binary relation for each mediator type m .

2.5 Transitivity

Freebase does not provide the transitive closure of transitive relations. Given R_1 and R_2 , the tuples of two relations r_1 and r_2 , we compute the transitive closure of tuples to be added during extraction as follows:

$$R_t = R_1 \circ R_2^+$$

Where R_2^+ is the transitive of relation r_2 and \circ is relation composition. This allows computing the transitive closure over two relations, e.g., *profession* and *is-specialization-of* to ensure that a person with the profession *physicist* also has the profession *scientist* (because the profession *physicist* is a specialization of *scientist*). The classic transitive closure is a special case where r_1 equals r_2 . We currently provide a manually compiled list of relations for which the transitive closure should be computed.

2.6 Duplicate Classes

A common problem in knowledge bases is that of duplicate entities or classes, often with identical or slightly different names. We follow a simple approach and merge two classes if they have the same name, ignoring case, and if the instances of one class are included in the other by a threshold. Let I_A and I_B be the instances/entities of some class A and B , respectively. We only merge class A into class B if the

⁵We always have $k \geq 1$ and for few relations, like “sibling”, we indeed have $k = 1$.

instances of class A are contained to at least 70% in class B , that is when:

$$\frac{|I_A \cap I_B|}{|I_A|} \geq 0.7$$

and vice versa.

3. USER INTERFACE

We provide a convenient user interface for performing complex searches on our version of the Freebase dataset, as described in the previous section. The main features are as follows. We encourage the reader to try our demo under <http://freebase-easy.cs.uni-freiburg.de>.

- (1) A single input field, as in standard text search.
- (2) Incremental query construction with suggestions (for matching types, instance and relations) after each keystroke.
- (3) Example tooltips for each relation (shown on mouse over), to help understand what the relation is about (relation names in Freebase are sometimes opaque).
- (4) Visual editing of the current query graph (e.g., removing a part or double-clicking a node to make it the new root).
- (5) Meaningful names (following Section 2.3) and images (loaded from Freebase, if available).
- (6) Proper ranking of results, using the scores from Section 2.2 where appropriate.
- (7) Sort by an arbitrary query element, i.p. dates and values.
- (8) Interactive query times, using the index from [1].

4. CONCLUSION

We provide a curated version of the Freebase data set that fixes several major usability issues with the original data set. We also provide a convenient user interface for interactive search and exploration, making good use of the various features we added. Several of the problems we addressed are major research problems in their own right. The solutions we provided here are simple and effective, yet by no means perfect. For example, our entity scores (derived from counts in the ClueWeb’12 corpus) work very well to bring the prominent entities to the top, but in some cases show an undesirable topic drift (e.g., Celine Dion is the fourth most prominent person). Our canonical entity names work like a charm for the more frequent entities, while names like *Berlin (United States) #2* could be improved.

5. REFERENCES

- [1] H. Bast and B. Buchhold. An index for efficient semantic full-text search. In *CIKM*, pages 369–378, 2013.
- [2] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.
- [3] E. Gabrilovich, M. Ringgaard, and A. Subramanya. FACC1: Freebase annotation of ClueWeb corpora, Version 1. (Release date 2013-06-26, Format version 1, Correction level 0).
- [4] E. Kaufmann and A. Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? In *ISWC*, pages 281–294, 2007.
- [5] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.

Semantic Full-text Search with Broccoli

Hannah Bast, Florian Baurle, Björn Buchhold, Elmar Haußmann

Department of Computer Science

University of Freiburg

79110 Freiburg, Germany

{bast, baeurle, buchhold, haussmann}@informatik.uni-freiburg.de

ABSTRACT

We combine search in triple stores with full-text search into what we call *semantic full-text search*. We provide a fully functional web application that allows the incremental construction of complex queries on the English Wikipedia combined with the facts from Freebase. The user is guided by context-sensitive suggestions of matching words, instances, classes, and relations after each keystroke. We also provide a powerful API, which may be used for research tasks or as a back end, e.g., for a question answering system. Our web application and public API are available under <http://broccoli.cs.uni-freiburg.de>.

1. INTRODUCTION

Knowledge is available in electronic form in two main representations: as natural language text (e.g., Wikipedia), and in structured form (e.g., Freebase). The central motivation behind our system is that both representations have their advantages and should be combined for high-quality semantic search.¹

For example, consider the query *Plants with edible leaves and rich in vitamin C*. Information about which plant contains how much vitamin C is naturally represented as fact triples. Indeed, this information is found in a knowledge base like Freebase. Information about which plants have edible leaves is more likely to be mentioned in natural language text. It is mentioned many times in Wikipedia, but we don't find it in Freebase (or any other knowledge base that we know of). In principle, the information could be added, but there will always be specific or recent information only described as text.

In the following, we describe how we combine these two information sources in a deep way. Figure 1 shows a screenshot of our demo system in action for our example query.

¹As a matter of fact, Wikipedia also contains some structured data, and Freebase also contains natural language text.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).

SIGIR'14, July 6–11, 2014, Gold Coast, Queensland, Australia.

ACM 978-1-4503-2257-7/14/07.

<http://dx.doi.org/10.1145/2600428.2611186>.

2. SYSTEM OVERVIEW

Preprocessing In principle, our system works for any given text corpus and ontology. For our demo we use the English Wikipedia (text) + Freebase (ontology). We preprocess this data in three phases. First, we link entities from the ontology to mentions in the full text, utilizing Wikipedia links and a set of heuristics as described in [1]. This provides the basis for our *occurs-with* operator explained below. Second, the full text is split into *contexts* that "semantically belong together" as described in [3]. This is key for results of high quality. Third, the special-purpose index described in [2] is built. This is key for providing results and suggestions in real time.

Queries The user interface allows to incrementally construct basic tree-like SPARQL queries, extended by an additional relation *occurs-with*. This relation allows to specify the co-occurrence of entities from the ontology with words from the text. For our example query, the back end computes all occurrences of plants that occur in the same context (see above) as the words *edible* and *leaves*. We also provide the special relation *has-occurrence-of*, to search for documents in which words and entities co-occur. Regular full-text search is thus included as a special case.

Query Suggestions Based on the input of a user, our system gives context-sensitive suggestions for words, classes, instances, and relations. The displayed suggestions always lead to hits, and the more / higher-scored hits they lead to, the higher they are ranked. This is an elementary feature for any system that utilizes a very large ontology. Without good suggestions it is very hard to guess how entities and relations are named, even for expert users.

Excerpts For each result hit (an entity or a document), matching evidence for each part of the query is provided. This is invaluable for the user to judge whether a hit indeed matches the query intent. The UI also provides (on click) detailed information about the NLP preprocessing (see above).

3. TARGET USERS

We see two uses of our system, and hence also two target groups of users.

Our first target group is expert searchers. Our search interface is more complex than ordinary keyword search or asking questions in natural language. The benefit is a powerful query language with precise result semantics. Under these constraints, we have made the query process as easy-to-use as possible. For example, there are tens of thousands of Wikipedia List pages like "Plants with edible leaves".

Words

Classes:

Food (9)
Ingredient (9)
1 - 2 of 37

Instances:

Broccoli (9)
Garden cress (4)
Kohlrabi (3)
1 - 3 of 9

Relations:

occurs-with
Compatible with dietary restriction (9)
Energy per 100g (9)
1 - 3 of 22

Your Query:

Plant *
├── occurs-with *
│ ├── edible leaves *
│ └── Vitamin C per 100g in mg *
│ └── > 50 *

Hits: 1 - 2 of 9


Broccoli

Ontology fact

Broccoli: is a **plant**; Vitamin C per 100g in mg **89.2**

Document: [Edible plant stem](#)

The **edible** portions of **Broccoli** are ... the **leaves**.



Garden cress

Ontology fact

Cabbage: is a **plant**; Vitamin C per 100g in mg **69**

Document: [Cress](#)

Plants cultivated for their **edible leaves** : **Garden cress** ...

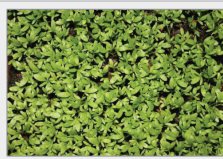


Figure 1: A screenshot of our demo system. The current query is visualized on the top right as a tree. Below, the result hits are shown, grouped by instance (entity) and ranked by relevance, together with evidence from the ontology and the full text. The query can be extended further with the search field on the top left. The four boxes below provide context-sensitive suggestions that depend on the current focus in the query, here: matching sub and super classes, instances, and relations for plants matching the current query.

Many of these are actually non-trivial semantic queries, which are hard to answer with traditional tools, like Google web search. We expect our tool to be a great asset for contributors to such List pages. We expect a similar benefit for expert searches in other areas, e.g., news (presidential campaign backers) or medicine (symptoms of a disease).

Our second target group is researchers in semantic search or engineers of such systems. They may want to use our system to explore the given data and thus gain insight into which facts are expressed in which ways. Engineers may also use our API as a back end for a more simplistic front end, suited for non-expert users. As a first step towards such a front end, we have integrated the following feature in our demo: when typing a query with three or more words without following any of the suggestions, the system tries to convert these keywords into a matching structured query. For example, try *mafia films directed by francis coppola*.

4. RELATED WORK

We see three lines of research closely related to our system.

First, we already mentioned systems for semantic search with more elaborate front ends. In particular, such allowing natural language queries like IBM’s well-known Watson [4], or standard keyword queries like in ad-hoc entity search [5]. When they work, such more intuitive front ends are clearly to be preferred. However, semantic search is complex and hard, and queries often fail. Then simple front ends lack the feedback needed to understand what went wrong and what can be done to ask a better query.

Second, there are various extensions of ontology search by a free-text component. A good example is the MQL language (similar to the more standard SPARQL) provided by Freebase (<http://www.freebase.com/query>). In MQL, objects of triples can also be string literals and these can be

matched against regular expressions and keyword queries. For example, find all songs containing the words *love* and *you* in their title. In principle, this could be used to simulate our *occurs-with* operator, but only very inefficiently; see [2, Section 4 and Table 1].

Third, information extraction (IE) aims at extracting factual knowledge from text. If this succeeded perfectly, ontology search would be all we need. There are two caveats, however. Whatever information was not extracted properly is lost. In our system, all the original information is kept and is, in principle, accessible by an appropriate query. Also, IE triples often have string literals as objects. Dealing efficiently with these requires a special index data structure, like the one behind our search.

5. REFERENCES

- [1] H. Bast, F. Bärle, B. Buchhold, and E. Haussmann. Broccoli: Semantic full-text search at your fingertips. *CoRR*, abs/1207.2615, 2012.
- [2] H. Bast and B. Buchhold. An index for efficient semantic full-text search. In *CIKM*, pages 369–378, 2013.
- [3] H. Bast and E. Haussmann. Open information extraction via contextual sentence decomposition. In *ICSC*, pages 154–159, 2013.
- [4] D. A. Ferrucci, E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefter, and C. A. Welty. Building watson: An overview of the DeepQA project. *AI Magazine*, 31(3):59–79, 2010.
- [5] J. Pound, P. Mika, and H. Zaragoza. Ad-hoc object retrieval in the web of data. In *WWW*, pages 771–780, 2010.

1266

90

Relevance Scores for Triples from Type-Like Relations

Hannah Bast, Björn Buchhold, Elmar Haussmann
Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
{bast, buchhold, haussmann}@cs.uni-freiburg.de

ABSTRACT

We compute and evaluate relevance scores for knowledge-base triples from type-like relations. Such a score measures the degree to which an entity “belongs” to a type. For example, Quentin Tarantino has various professions, including Film Director, Screenwriter, and Actor. The first two would get a high score in our setting, because those are his main professions. The third would get a low score, because he mostly had cameo appearances in his own movies. Such scores are essential in the ranking for entity queries, e.g. “american actors” or “quentin tarantino professions”. These scores are different from scores for “correctness” or “accuracy” (all three professions above are correct and accurate).

We propose a variety of algorithms to compute these scores. For our evaluation we designed a new benchmark, which includes a ground truth based on about 14K human judgments obtained via crowdsourcing. Inter-judge agreement is slightly over 90%. Existing approaches from the literature give results far from the optimum. Our best algorithms achieve an agreement of about 80% with the ground truth.

1. INTRODUCTION

Knowledge bases allow queries with a well-defined result set. For example, we can easily formulate a precise query that gives us a list of all *american actors* in a knowledge base. Note the fundamental difference to full-text search, where keyword queries are only approximations of the actual search intent, and thus result lists are typically a mix of relevant and irrelevant hits.

But even for well-defined result sets, a ranking of the results is often desirable. One reason is similar as in full-text search: when the set of relevant hits is very large, we want the most “interesting” results first. But even if the result set is small, an ordering often makes sense. We give two examples. The numbers refer to Freebase [7], the largest general-purpose knowledge base to date.

Example 1 (american actors): Consider the query that returns all entities that have *Actor* as their profession and *American* as their nationality. On our Freebase dataset, this query has 64,757 matches. A straightforward ranking would be by popularity, as measured, e.g., by counting the number of occurrences of each en-

tity in a reference text corpus. Doing that, the top-5 results for our query look as follows (the first item means the younger Bush):

George Bush, Hillary Clinton, Tim Burton, Lady Gaga, Johnny Depp

All five of these are indeed listed as actors in Freebase. This is correct in the sense that each of them appeared in a number of movies, and be it only in documentary movies as themselves or in short cameo roles. However, Bush and Clinton are known as politicians, Burton is known as a film director, and Lady Gaga as a musician. Only Johnny Depp, number five in the list above, is primarily an actor. He should definitely be ranked before the other four.

Example 2 (professions of a single person): Consider all professions by Arnold Schwarzenegger. Our version of Freebase lists 10 entries:

Actor, Athlete, Bodybuilder, Businessperson, Entrepreneur, Film Producer, Investor, Politician, Television Director, Writer

Again, all of them are correct in a sense. For this query, ranking by “popularity” (of the professions) makes even less sense than for the query from Example 1. Rather, we would like to have the “main” professions of that particular person at the top. For Arnold Schwarzenegger that would be: *Actor, Politician, Bodybuilder*. Note how we have a very-ill defined task here. It is debatable whether Arnold Schwarzenegger is more of an actor or more of a politician. But he is certainly more of an actor than a writer.

1.1 Problem definition

In this paper, we address the following problem, which addresses the issues behind both of the examples above.

Definition: Given a type-like relation from a knowledge base, for each triple from that relation compute a score from $[0, 1]$ that measures the degree to which the subject belongs to the respective type (expressed by the predicate and object). In the remainder of this paper we often refer to these scores simply as *triple scores*.

Here are four example scores, related to the queries above:

<i>Tim Burton has-profession Actor</i>	0.3
<i>Tim Burton has-profession Director</i>	1.0
<i>Johnny Depp has-profession Actor</i>	1.0
<i>Arnold Schwarzenegger has-profession Actor</i>	0.6

An alternative, more intuitive way of expressing this notion of “degree” is: how “surprised” would we be to see *Actor* in a list of professions of Johnny Depp. We use this formulation in our crowdsourcing task when acquiring human judgments.

Note that the “degree” in the definition above is inherently ill-defined, much like “relevance” in full-text search. In particular, different users might have different opinions on the correct “degree” (for example, on the four scores from above). However, it is one of the results of our crowdsourcing-based experiments that there is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SIGIR '15, August 09 - 13, 2015, Santiago, Chile.
© 2015 ACM. ISBN 978-1-4503-3621-5/15/08 ...\$15.00.
DOI: <http://dx.doi.org/10.1145/2766462.2767734>.

broad general consensus. Note that we do not take user preferences into account in this paper; we consider this an interesting topic on its own.

1.2 Variants and related problems

The reader may wonder about variants of the problem definition above, and related problems from the literature. We briefly discuss these here.

Degree of Belonging vs. Correctness vs. Accuracy

Our scores are different from correctness or confidence scores. For example, in knowledge fusion, multiple sources might provide conflicting information on the birth date of a person. In those situations we need assessments on the confidence of a particular information and which information is probably correct [12].

Our scores are also different from scores that measure the accuracy. For example, an estimate for the population of a country might be off by a certain margin but not completely wrong, or only a range for the value is given.

We consider a single knowledge base with non-conflicting and precise information. Indeed, for a knowledge base like Freebase, over 99% of all triples are reported to be correct [7]. Even if there are incorrect triples, it is a pleasant side effect of our framework that they would get low scores.

Graded scores vs. Binary scores

We started our research on this topic with binary scores, which we called *primary* and *secondary*, in accordance with existing research on graded relevance in full-text search.¹ We later found that both our crowdsourcing experiments and our algorithms naturally provide finer grades that also make sense in practice. We hence use a continuous score in our definition above. The range $[0, 1]$ is merely a matter of convention.

In our crowdsourcing tasks in Section 3 we still ask each judge for only a binary judgment. However, aggregation of multiple judgments for each triple leads again to a graded score.

Which kind of relations

In principle, our algorithms work for all relations. However, we found that for relations other than type-like, the kind of scores we are interested in are either trivial or can be obtained using other, simpler methods. We distinguish three types of relations.

- Functional relations like *birth date* or *weight*, where each subject has exactly one object. Since we assume a single knowledge base where (almost) all triples are correct, we could simply assign a score of 1.0 to all such triples.
- Non-functional relations between two concrete entities, like *acquired* (between two companies) or *acted in* (between an actor and a movie). For such triples there are two simpler options for a good relevance score, which we do not explore further in this paper. The first option is to take the value from another relation of the knowledge base, like the date of the acquisition or the rank of the actor in the cast listing. This relation could be assigned manually (the number of relations in a knowledge base is relatively small). Or it could be determined heuristically (interesting, but technically completely different from the methods we present). The second option is to adapt our method for type-like relations algorithms also for these relations. The main challenge for finding witnesses in the full text is then to recognize variations of the (fixed) predicate name and not of the (varying) object name. This makes our problem much easier; see Section 4.

¹We assume that all triples in our knowledge base are correct, so there are no triples expressing a relationship that is *not relevant*.

- Non-functional relations between an entity and an abstract group or type, like *profession* (between a person and a profession) or *nationality* (between a person and a nationality). We know of no knowledge base with explicit values for the “degree” to which an entity belongs to a certain type. In particular, this is true for *profession* and *nationality*. As we will see in Section 4, such relations are also the hardest for our approaches, because they require a separate classifier for each distinct type (e.g., each distinct profession and nationality), instead of just one for the whole relation. This is why we selected only type-like relations for our benchmark.

We further remark that type information is central to many entity queries. For example, all three tasks from the TREC 2011 Entity Track [2] ask for lists of entities of a particular type. Also, most of the entity queries currently supported by Google are for entities of a certain type, like in the two use cases from our introduction; presumably because of their popularity amongst users.

Triple Scores vs. Full Ranking

The reader may ask why we study scores for individual triples when the motivation is the ranking of results for entity queries. We give two main reasons.

First, the triple scores we consider are a crucial component of any such ranking, and the problem by itself is hard enough to warrant separate study. In fact, in Section 2 we discuss related works on ranking for entity queries that require such scores as input.

Second, for many popular entity queries (like our two use cases from the introduction), our triple scores are basically all that is needed to obtain a good ranking. For the first use case (“american actors”), ranking is a simple matter of combining our triple scores with a popularity measure for each entity. Popularity measures are easily obtained, e.g., by counting the number of mentions in a reference corpus. For the second use case (“professions of a single person”), triple scores are exactly what is needed. Note that for the second use case, a proper ranking is still missing on Google.

1.3 Basic Idea and Challenges

Our basic idea is to find “witnesses” for each triple in a given large text corpus. The more witnesses we find and the more significant they are, the larger the score for that triple.

For example, consider the *profession* relation. In a “learning” step, we compute words that are associated with each profession. For example, for the profession *Actor* these could be: actor, actress, film, cast. For each entity (e.g., *Johnny Depp*) we identify occurrences of these words semantically related to that entity in the text (e.g., *Paramount decided to cast Depp ...*). Then we compute weighted sums of the number of such occurrences for each possible profession. In a final step, we normalize these sums to obtain our scores. This sounds simple, but there are a number of challenges:

- Learning must be unsupervised, given that we have to learn different words for every different possible “type”. For example, there are more than 3,552 different professions in Freebase.
- When is a word “semantically related” to an entity in a text? It turns out that considering co-occurrence in the same sentence works, but that a deeper natural language processing helps.
- How much weight to give to each occurrence of a word? It turns out that results can be boosted by graded scores for different words.
- How to get from weighted sums to normalized scores? It turns out that a simple linear mapping works but is often not optimal.
- Why use text and not the knowledge base? For a long time, we also experimented with other facts from the knowledge base as complementary or even as the sole information source. For example, the fact that someone is or once was president of the U.S.

implies a high score for the profession Politician. However, we found this knowledge to be consistently less available (especially for less popular entities) and nowhere more reliable than full text. More details about this from our experiments in Section 5.4.

1.4 Contributions and Overview

We consider the following as our main contributions:

- We identify the computation of relevance scores for triples from type-like relations as an interesting research problem, which so far has not achieved much attention. These scores are essential for properly ranking many popular kinds of entity queries.
- We designed and make available a first benchmark for this problem. The required large number of relevance judgments (over 14K) were obtained via crowdsourcing. Given the hard-to-define nature of our scores, designing the task such that untrained human judges provide proper and consistent judgments was a challenging task on its own. Inter-judge agreement is about 90%, which confirms the soundness of our problem definition. See Section 3.
- We introduce a variety of methods (partly new, partly adapted) to compute these scores. All our methods work with an arbitrary knowledge base and text corpus. See Section 4.
- We implemented and evaluated all methods against three baselines, using Freebase as knowledge base and Wikipedia as text corpus. Our best methods achieve an agreement of about 80% with the ground truth. We considered many variants and ideas for improvement, none of which gave better results. See Section 5.
- We make our evaluation benchmark as well as all our code publicly available (see Section 5). In particular, this allows full reproduction of our results.

2. RELATED WORK

There exists a large amount of work about ranking and other problems based on the kind of relevance scores we study here. However, in all these works such scores are assumed to be given. We know of no work that addresses how to compute these scores in the first place. We give a short overview of the various approaches.

Triple Scores for Ranking Structured Queries Several pieces of work deal with integrating scores in a framework for ranking structured queries. For example, in [8], the authors propose an extension to RDF they call Ranked RDF, [13] proposes a ranking model for SPARQL queries with possible text extensions based on Language Models, and [11] discusses how to combine several kinds of scores associated with triples into a meaningful ranking. In all these frameworks, scores that are similar to our triple scores are assumed to be given.

Fuzzy Databases / Knowledge bases This is an old and well-established research area, considering all kinds of “fuzziness” in an information system, including: uncertainty, imprecision, and fractional degrees of membership in sets; see [20] for a survey. However, this body of work is almost entirely about modeling this fuzziness, and how to solve standard tasks (like queries or reasoning) based on fuzzy information. The fuzzy information itself is, again, assumed to be given. We know of no work in this area, where fuzzy scores of the kind we consider are actually computed.

Ranking for Relational Databases SQL allows explicit ordering of the query results (using the *order by* keyword). Still, there are many meaningful queries to databases that return a huge number of results. When confronting users with those results, ranking plays an important role. This problem is tackled in [9]. Apart from the many-answers problem, ranking for databases becomes important when adding a keyword-search component. BANKS [5] has

the goal of ranking possible interpretations of a keyword query. Similarly, ObjectRank [1] also takes keyword queries, but ranks “database objects” (e.g., a paper, an author, etc.) according to a query. These approaches share the fact that they work exclusively on the data (or knowledge) base. Going from there, they try to use the structure induced by foreign keys to provide a ranking. For our specialized use case, this is not well suited. Even in a perfect world, where such an approach is able to find the perfect connections to influence the ranking (a hard task in itself), it is restricted to data in the knowledge base. However, as discussed in Section 1.3, we have found structured knowledge to be less available than that from full text for computing our relevance scores.

Ranking Semantic Web Resources In the Semantic Web, one is confronted with data from many different sources, and of highly varying quality. This gives rise to the problem of ranking these data sources with respect to a given query or topic. For example, TripleRank [16] achieves this by representing the Semantic Web as a 3D tensor and then performs a tensor decomposition. Despite the related-sounding name, this is very different from our scenario, which is about ranking of entities from a single knowledge base.

Topic Models On a technical level, some of our methods are related to *topic models*. These derive high-level topics of documents by inferring important words for each topic and the topic distribution for each document. In our setting, a document could be seen as a subject (e.g., person) and topics as different types (e.g., her professions). One state-of-the-art unsupervised topic model is Latent Dirichlet Allocation (LDA) [6]. LDA (and related methods) infer topics given only the text as input. In a supervised extension called Labeled LDA (L-LDA) [21], topic labels (e.g., our professions) can be provided for each document as part of the input. We compare our methods against L-LDA in Section 5.

3. ESTABLISHING A BENCHMARK

Ranking problems and the notion of relevance are inherently subjective and vague. Human relevance judgments are not only necessary to evaluate algorithms that attempt to solve the problem, but also help understanding it. This has happened for keyword search, where ranking became an established problem with a universally shared understanding.

We decided to use crowdsourcing to create a suitable benchmark for our triple scores for exactly these two reasons: evaluation and problem refinement. Since this helps to understand the problem at hand, we discuss our benchmark before we describe our algorithms to solve the problem.

Recall our use case example from Section 1. We want humans to judge to what extent a person has a certain profession. In this section, we discuss how we select a representative sample of entities, present two ways to set up the task, and explain why one is superior. We then analyze the result of the crowdsourcing experiment. The ground truth files are available for download together with all reproducibility material under the URL given in Section 5.

3.1 Selecting Persons for Evaluation

Our benchmark should feature a representative sample of persons that contains all levels of popularity. This is important for two reasons. First, it can be expected that more information is available for popular persons, so that some approaches might work better (or even only) for those. Second, as discussed in the motivation, while the ranking problem we address also exists for less popular persons, it is more pronounced for popular persons.

Selecting persons from Freebase with more than one profession leaves us with about 240K persons. We use the number of times

a person is mentioned in Wikipedia to approximate popularity and restrict our choice to persons having Wikipedia articles. This has practical reasons. In principle, any text collection could be used, but Wikipedia is easy to obtain and, due to hyperlinks, no hard Entity Linking problem has to be solved. Apart from that, we can point human judges directly to the Wikipedia article and hence enable them to make an informed decision without much effort. We observe that (as could be expected) there are lots of people with none or very few mentions and few people with lots of mentions (up to almost 100K mentions). Therefore, a random sampling would almost exclusively contain unpopular entities.

We define buckets of popularity and take a uniform random sample from each bucket. The buckets used were for a popularity (number of mentions) $< 2^4$, between 2^i and 2^{i+1} for $4 \leq i \leq 13$, and $\geq 2^{14}$. In total, we took about 25 samples from each of the 12 buckets. This left us with 298 persons or a total of 1225 person-profession tuples that nicely cover different levels of popularity.

3.2 Evaluate a Single Profession of a Person

An obvious approach is to present a person and his or her profession to a judge and ask whether the profession is primary or secondary for that person. For example:

Arnold Schwarzenegger and the profession *Bodybuilder*. Is the profession primary or secondary for this person?

The task was enriched with instructions, including definitions of the factors listed above. However, this definition is extremely hard to communicate precisely. Also, there is a lot of subjectivity involved: what “feels” primary to one judge does not to another.

Eventually, it turned out that judges mostly resorted to the following strategy: label the profession that is first mentioned in Wikipedia as primary, and all others as secondary. Indeed this is one of our simple baselines in Section 5. Obviously, this cannot work for persons with more than one primary profession.

3.3 Evaluate All Professions of a Person

An improved version that turned out to work well is the following. Instead of labeling a single profession of a person, we ask to label all professions of a person. Additionally, we provide simpler instructions but enrich them with a set of diverse examples of labeled persons. An example is depicted in Figure 1. All professions of *Barack Obama* must be dragged into the box for primary or secondary professions.

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 1

Lawyer 1

Law Professor 1

Author 1

Writer 1

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

Figure 1: Condensed illustration of the crowdsourcing task. All professions must be dragged into the box for primary or secondary professions. For the complete version including instructions and examples see, go the URL from Section 5.

Below that, we show four examples illustrating a diverse set of possible labelings: Michael Jackson (many professions, some primary some secondary), Ronald Reagan (many professions, two of them primary), Mike Tyson (three professions, only one primary), James Cook (two professions, both primary).

This worked well, and we assume this is mainly due to two reasons:

- (1) People could do research on the person, and thus judge all the professions of that person in relation to each other.
- (2) The diverse examples helped to form an intuitive theory of what we expected.

3.4 Crowdsourcing Results

We had the professions of each person labeled by seven different judges. As final score for each triple we sum up the number of primary labels, which is then in the range of 0 (all judges label secondary) to 7 (all judges label primary).

On the 298 persons (1225 person-profession tuples) the inter-annotator agreement according to Fleiss’ Kappa [15] is 0.47 for the binary judgments. This can be considered moderate agreement. However, what our experiment actually does is to determine a hidden probability p with which a human judge finds a person’s profession to be primary. In fact, we use a binary experiment to get gradual scores. Note that, for certain triples, the desired result may actually be a probability around 0.5. Measuring inter-annotator agreement does such a scenario no justice. Instead, we want to make sure that (1) all judges base their decision on the task and the data, and (2) the obtained probabilities are significantly different from random decisions.

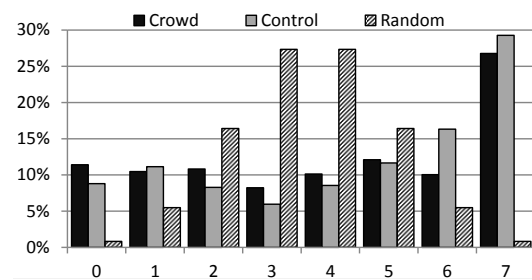


Figure 2: Histogram of score distribution of our crowdsourcing task, the control run, and expected results for randomly (with $p = 0.5$) guessing judges (rounded).

For (1), we have run the same task again (with different annotators) on a subset of every third person (386 person-profession tuples) from our initial sample. Figure 2 depicts how much the scores differ between this control run (*control*) and the full run (*crowd*).

For (2), looking at the distribution of scores shows that they are very far from a random decision. Figure 2 shows a distribution that prefers border cases (especially primary professions) and contains only reasonably few triples with average score (3 or 4). In contrast, a random or unclear task would lead to a distribution with mostly average scores and rare border cases.

In Section 5, we report more details on the results of the control run. The human judges based 95% of their decisions only on the Wikipedia page of the respective person. As we report in Section 5.4, our automatic methods perform much worse when we restrict them in this way. This shows how hard the judgment task is: the relative relevance of the various professions mentioned on a person’s Wikipedia page is often highly implicit in the text.

4. COMPUTING TRIPLE SCORES

We propose a variety of algorithms to compute triple scores. For illustration purposes we describe the algorithms using the *profession* relation of a person, but the algorithms are not specific to that relation and should work with any type-like relation. In Section 5 we show that they are equally effective to derive scores for nationalities of persons. Likewise, we use Wikipedia as our source of text and Freebase as knowledge base, but the approaches are not specific to that data. None of our algorithms requires manually labeled data. Instead, we make use of existing facts in an unsupervised or distantly supervised fashion. The crowdsourcing judgments are only used for evaluation.

The different algorithms provide different output: binary classifications, weighted sums and probabilities. In Section 4.6, we describe how we map their output to the range $[0, 1]$.

4.1 Training Data

We want to avoid manually labeled training data. Instead, we use the following definition to obtain positive and negative training examples for each profession:

Positive: people with only the given profession or any specialization of it, i.e. those for which the profession has to be primary.
Negative: people that do not have that profession at all

Thus, Humphrey Bogart is a positive example for the profession *Actor* (because this is his only profession according to Freebase) and Barack Obama, who is listed as politician, lawyer and more (but not as actor), is a negative example for that profession.

We also experimented with other criteria (e.g., allowing the persons used as *positive* examples to also have other professions) but found this selection to work best across all approaches.

4.2 Selecting Text for each Person

All approaches analyze text about a person to derive triple scores. As text we use the English Wikipedia², utilizing the fact that persons in Freebase often have a link to their Wikipedia article. We have pre-processed Wikipedia and performed Entity Recognition and Linking, as well as anaphora resolution as described in [3]. With each person we associate all words that co-occur with a linked mention of the person in the same *semantic context*, as provided by the pre-processing. Each semantic context is a sub-sequence of the containing sentence that expresses one fact from the sentence. This yields slight improvements over using full sentences (where numerous facts can be mixed) as contexts. Further, we have found that stemming has almost no effect. These and other variations are discussed in more detail in Section 5.4.

4.3 Binary Classifier per Profession

We train a binary classifier to decide whether a profession is primary or secondary for a given person. For each person we extract features from her associated text. We use word counts as features, which we normalize by the total number of word occurrences in the person’s associated text. Thus, feature values are between 0 and 1 (much closer to 0). Weighting feature values by their tf-idf score (instead of the normalization) had no positive effect.

Before training the classifier, we balance positive and negative training examples by randomly selecting an appropriate subset of the negative training examples (there are always more negative than positive training examples).

A logistic regression classifier with L2-regularization is then trained on the balanced training data. As a linear model, logistic regression has the benefit of an intuitive interpretation of features

²downloaded in August 2013

1. cast	28.21	5. directed	-6.64	9. university	-5.16
2. actor	12.46	6. starring	6.22	10. written	-4.97
3. actress	11.50	7. role	5.62	11. voiced	4.50
4. played	7.19	8. stars	5.26	12. actors	4.48

Table 1: Top features (positive and negative) and their weight learned by the logistic regression classifier for the profession *Actor*.

weights. Table 1 lists the features with top weights learned for profession *Actor*.

The learned feature weights form a model for each profession. These models can then be used for persons with multiple professions to make a binary decision if each of his or her professions is primary or not (secondary).

4.4 Count Profession Words

We want to mimic the behavior of humans that want to solve our problem. Therefore, we look at the text associated with an entity and find out how big the portion of text is that discusses profession matters. In the simplest way, we could count how often a profession is mentioned in text associated with the person.

We cannot count exact occurrences, however, because of professions that consist of more than one word, e.g., Film Score Composer. Looking for mentions of such professions already puts them at a disadvantage (or advantage depending on full vs. partial match counting) compared to one-word professions like Composer. Additionally, often slight variations of a profession are mentioned, e.g., actor and actress.

To overcome this issue, we define a suitable prefix for each profession. We can do this automatically (using an off-the-shelf stemmer and the longest common prefix of the stem and the original word) or manually once for each profession. In our experiments (Section 5), we compare manually chosen prefixes as a strong baseline.

Besides the profession names themselves (or their prefixes), we have found that there are many other words that indicate, that some part of the text is about a given profession. For example, consider the positive features learned by the binary classifier as presented in Table 1. We extend our counting based approach by automatically computing indicator words in the following way: For each profession, take all words in the associated text for persons in the *positive* training data (see Section 4.1), compute their tf-idf value and rank the words by that value. During prediction, we set the weight of an indicator mention to $1/\text{rank}$ and build the sum over the weights of all indicator mentions.

1. cast	1.00	5. role	0.20	9. television	0.11
2. actor	0.50	6. starring	0.16	10. appeared	0.10
3. actress	0.30	7. played	0.14	11. born	0.09
4. film	0.25	8. best	0.13	12. series	0.08

Table 2: Top words (by tf-idf) and their weight ($1/\text{rank}$) for the profession *Actor*.

Table 2 shows the top words and their weights for the profession *Actor*. There is high overlap with the top features learned by the binary classifier shown in Table 1. Note, that the word weights were computed only from the text associated with *positive* training examples and idf values based on the whole corpus. The *negative* examples were not needed.

4.5 Generative model

We formulate a generative model where each person is associated with text that can be generated as follows. For a person with k professions and n word occurrences in the associated text: Pick one of the k professions with probability $P(p_i)$; generate word w_j with $P(w_j|p_i)$; repeat until all n words are generated. The joint probability for word w and profession p is then $P(w, p) = P(w|p)P(p)$.

Note that for a given text, the professions selected in the first step above are unobserved. We would like to infer those, because, intuitively, they represent the amount of text that can be associated with a certain profession. We derive these using a maximum likelihood estimate.

Let tf_j be the term frequency of word j , $P(p_i)$ be the probability of profession i and let $P(w_j|p_i)$ be the probability for word j under profession i . The log-likelihood of producing text consisting of n words for k professions is:

$$\log \mathcal{L} = \sum_{j=1}^N [tf_j \cdot \log \sum_{i=1}^k (P(w_j|p_i)P(p_i))]$$

Training We use the *positive* training examples to derive word probabilities $P(w|p)$. We distinguish two ways. (1) use the term frequency of terms in the text associated with all training examples and assign probabilities accordingly. (2) use the tf-idf values as term frequencies to avoid common, unspecific terms with high probabilities. When using tf-idf values, we use them for both: $P(w|p)$ and as new tf_j when calculating LP . For efficiency reasons, we only keep the probabilities for the top 30K words. We have found that tf-idf values work better and restrict our examples to this setup for the remainder of the paper.

Text that is associated with a person often contains information that is not related to professions, e.g., family and personal life. Therefore, we add a pseudo profession to each person in order to account for these words. We use all text in the English Wikipedia to calculate $P(w|p)$ for the pseudo profession.

1. cast	0.023	5. role	0.007	9. appeared	0.004
2. actor	0.009	6. starring	0.005	10. television	0.004
3. actress	0.008	7. played	0.005	11. born	0.004
4. film	0.008	8. best	0.004	12. roles	0.004

Table 3: Top word probabilities for the profession *Actor*.

Naturally, the top word probabilities depicted in Table 3 are in line with the top word weights presented in Table 2. This is not surprising, since both values are based on the tf-idf scores of words in text associated with the *positive* training samples. The probabilities, however, do not differ as much as the weights we have assigned for the counting approach.

Prediction To derive profession probabilities \vec{p} , which have to sum to 1, we maximize the log-likelihood:

$$\vec{p} = \arg \max_{\vec{p}} \sum_{j=1}^N [tf_j \cdot \log \sum_{i=1}^k (P(w_j|p_i)P(p_i))], \text{ s.t. } \sum_{i=1}^k p_i = 1$$

To find the maximum likelihood estimate we use Expectation Maximization [10]. Similar to the generative model for pLSI [17] we treat the topic (profession) assignments to word-occurrences as hidden variables. EM iteratively computes posterior probabilities given the hidden variables in the expectation step (E) and then maximizes parameters for the previously computed probabilities in the maximization step (M). The E and M steps are identical to the steps in [17], with the difference that we treat $P(w|p)$ as fixed, because

we already computed those from the *positive* examples as described above.

4.6 Mapping to Triple Scores

The above approaches yield a variety of results. Binary classifications, weighted sums and probabilities. However, we actually want to compute scores for triples. While this is trivial for binary classifications (assign minimum and maximum), we distinguish two approaches for sums and probabilities. Keep in mind that we assume there is at least one primary profession for each person. For comparison with the crowdsourcing judgments, we want scores from 0 to 7 but the methods apply for any desired magnitude and, without rounding, naturally allow continuous scores as well.

Maplin Linearly scale computed values to the range 0 to 7. In practice, just divide all sums or probabilities by the highest one. Then multiply by 7 and round to the nearest integer.

Maplog Scale computed values to the range 0 to 7 such that the next highest score corresponds to twice the sum or probability. In practice, divide all sums or probabilities by the highest one. Then multiply by 2^7 , take the logarithm to base 2, and finally take the integer portion of the result.

We have found that the intuitive *Maplin* works much better with the sums of weights obtained by counting triples. *Maplog*, however, is stronger when mapping probabilities to triples scores. This is true for both, our generative model, and the topic model, Labeled LDA, we compare against. The probabilistic models tend to assign high probabilities to the top professions, leaving small probabilities to all others. Differences between debatable professions and definite secondary are small in total probability difference value but still observable when comparing magnitudes.

5. EVALUATION

We first discuss the experimental setup: data + ground truth, algorithms compared and quality measures. In Section 5.2, we present and discuss our main results. In Section 5.4, we discuss many variants (algorithms and quality measures) that we also experimented with but which did not turn out to work well.

All of the data and code needed to reproduce our experiments are available under <http://ad.informatik.uni-freiburg.de/publications>.

5.1 Experimental Setup

Data We extracted all triples from the Freebase relations *Profession* and *Country of nationality*. In all experiments we only consider persons with at least two different profession or nationalities, i.e., we only consider the non-trivial cases. Files with all these triples are available under the above URL.

Ground truth For the *profession* relation, we randomly selected a total of 1225 triples pertaining to 298 different people entities, as described in Section 3.1. Each of these triples was then labeled independently by 7 human judges, using crowdsourcing, as described in Section 3.1. This gives a total of 8.575 labels. We repeated the task for all 386 triples of a random subset of 98 from the 298 people. This gives us another 2.702 labels. We used these as control labels to assess the quality of our main set above; see Section 5.2 below. We have presented the distribution of the scores in the ground truth in Section 3.4 in Figure 2. For the *nationality* relation, we randomly selected a total of 162 triples pertaining to 77 different people entities and ran the same experiment. This gives us another 1134 judgments.

Algorithms Compared We compare the following: three baselines, six algorithms, and the output of two control experiments.

We normalized all approaches to yield an integer score from the range 0..7 for each triple.

First For each person, take the entity’s description from Freebase³. Look for the first literal mention of one of that person’s profession. That profession gets a score of 7, all other professions get a score of 0. This may look overly simplistic, but actually this is what most of the judges did in the previous version of our task as presented in Section 3.2.

Random Make 7 independent judgments for each triple, where for each judgment primary and secondary are equally likely. That is, pick the score at random from a binomial distribution with $n = 7$ and $p = 0.5$. This simulates human judges that guess randomly.

Prefixes For each triple, count the number of occurrences of a hand-picked prefix of the profession (same for all professions) in the Wikipedia article of the person. Map these counts to scores (per person) using *Maplin*, as described in Section 4.6.

Labeled LDA (LLDA) The closest approach from previous work, as described in Section 2. We use a topic label for each profession and label each person with all of her professions. The Dirichlet prior, α , is set to 2.0 and we run Gibbs sampling for 100 iterations. Other parameter settings for α gave much worse results and more iterations showed no improvement. We map the probabilities to scores using *Maplog*.

Words Classification For each triple, make a binary decision (score 0 or 7) using the logistic-regression classifier (see Section 4.3).

Words Counting For each profession, learn a weighted list of words, as described in Section 4.4. For each triple, add the weights of all profession words in all contexts containing the entity. Map these weight sums to scores using *Maplin*.

Words MLE For each triple, we derive a score using the generative model described in Section 4.5 and the *Maplog* mapping.

Counting Combined For each triple, use the *Words Counting* method. Additionally look at the decision of the *Words Classification* method. If the binary classification is positive (score 7), increase the score to the average of the two. The intuition behind this is, that for strongly related professions (e.g., Poet and Playwright), it is hard to attribute a text passage to one of the two. The binary decisions made by the *Words Classification* method do not have this problem.

MLE Combined Similar to *Counting Combined* but starting with the *Words MLE* method instead of the counting.

Control Judges Take the labels from the human control judges, as described above.

Control Expected The expected values when comparing two scorings, where in each scoring each of the seven judgments for a triple is primary with probability p , where p is chosen from the posterior distribution given score s , where s is the score given by the crowdsourcing judges.⁴

Evaluation measures We use two kinds of measures: score-based and rank-based.

Score-based The score-based measures directly compare the scores of two sequences s and s' of triple scores for the same sequence of n triples. We consider two measures: *accuracy* and *average score deviation*. The accuracy has an integer parameter δ and measures the percentage of triple scores that deviate by at most δ . That

³For most entities, this is just the abstract of the corresponding Wikipedia page.

⁴That is, $Pr(p = k/7) \propto Pr(X = k)$, for $k = 0..7$, where X is from the binomial distribution $B(7, s/7)$.

is, $Acc-\delta = |\{i : |s_i - s'_i| \leq \delta\}|$. The average score deviation is simply the average over all absolute score differences. That is, $ASD = \sum_i |s_i - s'_i|/n$. We will see that the relative performance of the various methods shows in both measures, but that the *Acc* figures are more intuitive and insightful.

Rank-based The rank-based measures compare two rankings of the same sequence of triples. We will use them to compare two rankings of all professions / nationalities of a single person, and the average over all persons. We consider three standard ranking measures: Kendall’s Tau, the footrule distance, and nDCG.

Because scores in the gold standard are discrete, items often share a rank. Such a ranking with ties constitutes a *partial ranking* [14]. To compare partial rankings we use adapted versions of Kendall’s τ from [14]: $\tau_p = \frac{1}{Z}(n_d + p \cdot n_t)$, where n_d is the number of *discordant* (inverted) pairs, n_t is the number of pairs that are tied in the gold standard but not in the predicted ranking or vice versa, p is a penalization factor for these pairs which we set to 0.5, and the normalization factor Z (the number of ordered pairs plus p times the number of tied pairs in the gold standard).

The Spearman footrule distance counts the displacements of all elements. It is $f = \frac{1}{Z} \sum |\sigma_p(i) - \sigma_g(i)|$, where i ranges over the items of the list, and $\sigma_p(i)$ and $\sigma_g(i)$ is the rank of item i in the predicted partial ranking and gold standard partial ranking, respectively. The normalization factor Z corresponds to the maximum possible distance and causes f to be in the interval $[0, 1]$.

For nDCG, we remark that it also takes the exact scores into account: ranking a triple lower than it should be is punished more, the higher the score of that triple is.

5.2 Main results

Score-based evaluation of the profession triples We first discuss the results for our score-based measures for the *profession* relation, shown in Table 4.

Method	Accuracy (Acc)			Average Score Diff
	$\delta = 1$	$\delta = 2$	$\delta = 4$	
First	41%	53%	71%	2.71
Random	31%	55%	91%	2.39
Words Classification	47%	61%	78%	2.09
Prefixes	50%	64%	83%	2.07
LLDA	50%	68%	89%	1.86
Words Counting	57%	75%	94%	1.61
Words MLE	57%	77%	95%	1.61
Count Combined	58%	77%	95%	1.52
MLE Combined	63%	80%	96%	1.57
Control Expected	75%	91%	99%	0.93
Control Judges	76%	94%	99%	0.92

Table 4: Accuracies and ASD for the *profession* relation, best methods last.

The last line (Control Judges) shows that the human judges rarely disagree by more than 2 on the 0-7 scale. The *Acc-4* measure shows that there are almost no stark disagreements, that is, by more than 4.⁵ A disagreement up to 1 is not unusual though. The average score deviation is 0.92. As the next to last line (Control Expected) shows, this is essentially what would be expected from random

⁵Note that such disagreements can only happen for scores 0-2 and 5-7.

fluctuation. Acc-2 hence seems to be the single most intuitive measure (for integer scores on a scale 0-7). We therefore emphasized the results for that measure in Table 4.

Our binary classification algorithm achieves an Acc-2 of 61%, which gradually improves to 80% for our most sophisticated algorithm. There is hardly a difference between the MLE approach and word counting with proper normalization. Our best approach makes glaring mistakes (Acc-4) for only 4% of all triples. For the Acc-2 measure, our best approach is still more than 10% away from the ideal. In Section 5.4, we discuss various options for improvement, none of which actually gives a significant further improvement though. We conclude that our problem is what could be called “NLP-hard”. Under that condition, we consider an Acc-2 of 80% a very good result.

The simple “First” baseline performs similarly bad as the “Random” baseline, which simulates a random guess for each judge. Note that the “Random” baseline makes glaring mistakes (Acc-4) only for 9% of the triples. This is because the scores of this baseline follow a binomial distribution, with a probability of 0.55 that the score is 3 or 4. For these two scores, the deviation from the “true” value cannot be larger than 4. For the more extreme scores, the probability of being off by more than 4 is also low. Acc-2, however, is only about 55% for “Random”. Note that a more extreme version of “Random”, which picks only one of the scores 3 or 4 at random for each triple, would achieve an Acc-4 of 100%, but an Acc-2 of only 52%.

The relative performance of the various approaches is reflected by the average score difference (ASD), shown in the last column of Table 4. The various Acc measures provide a more refined picture though.

Rank-based results for the *profession* triples We next discuss the results for our rank-based measures for the *profession* relation, shown in Table 5.

Method	Kendall	Footrule	nDCG
Random	0.51	0.58	0.80
First	0.40	0.47	0.92
LLDA	0.32	0.38	0.88
Words Classification	0.32	0.35	0.88
Prefixes	0.31	0.35	0.88
Words MLE	0.23	0.28	0.94
Words Counting	0.24	0.29	0.94
Counting Combined	0.24	0.28	0.94
MLE Combined	0.22	0.27	0.94
Control Judges	0.18	0.21	0.97

Table 5: Average rank differences for the *profession* relation, best methods last.

We observe that the relative quality of the various baselines and algorithms is about the same in all three measures. Since Kendall is the simplest and perhaps most intuitive measure, we have highlighted the results for that measure in Table 5.

Footrule is always slightly larger than Kendall. The nDCG values are relative large already for the simple Random baseline. This is an artifact of the short lists we are comparing here (for each person, his or her professions, which are often only two or three). Indeed, for a list with only two items, there are only two possible values for nDCG: 1 and $1/\log_2 3 \approx 0.63$. The average of the two is ≈ 0.82 .

The relative order of the methods is similar as for the score-based evaluation of Table 4. The only notable exception is that in the rank-based evaluation, the simple “First” baseline is significantly better than “Random”, while in the score-based evaluation, it was the other way round. The reason is simple: “First” almost always gets the first item in the list right (the first profession listed in Wikipedia, is almost always the most obvious profession of the person in question). For persons with only two professions, this already gives the perfect ranking. For persons with three professions, this already gets 2 of the 3 possibly transpositions right. In the score-based evaluation, the score for the triple with this “first” profession is just one out of k , when the person has k professions. Apart from that, “First” (and “Words Classification”) make binary decisions for one of the most extreme scores. This is punished by the Accuracy measures. Random, on the other hand, tends to select the middle ground scores 3 and 4 which are not punished as strongly.

Note that the compared rankings are invariant under score mapping (*Maplin* or *Maplog*, see Section 4.6). Hence, score mapping affects neither Kendall nor Footrule. It does, in principle, affect nDCG, since that measure also takes the actual scores into account. We found the effect to be insignificant though (at most 0.02 difference for affected approaches).

Score- and ranked-based evaluation of the *nationality* triples

We repeated the experiments above for the triples from the *nationality* relation. We focused on the baselines and the best-performing methods.

Method	Accuracy (Acc)			Average Score Diff
	$\delta = 1$	$\delta = 2$	$\delta = 4$	
First	28%	36%	51%	3.89
Random	31%	55%	91%	2.83
Prefixes	41%	52%	71%	2.21
Words MLE	58%	78%	95%	1.71
Words Counting	63%	78%	96%	1.34
Control Expected	76%	92%	99%	0.82

Method	Kendall	Footrule	nDCG
Random	0.51	0.58	0.80
First	0.41	0.42	0.90
Prefixes	0.51	0.53	0.88
Words MLE	0.44	0.45	0.92
Words Counting	0.36	0.37	0.94

Table 6: Score-based and ranked-based results for the *nationality* relation, best methods last.

Table 6 shows the results for our experiments with triples from the *nationality* relation. Sophisticated approaches have performance similar to the run on triples of the *profession* relation (see Tables 4 and 5). This is good news, because it was not guaranteed that Wikipedia texts provided a sufficiently strong signal for our algorithms to perform well. Interestingly, baselines “First” and “Prefixes” are weaker on this task. Apparently, direct mentions of a nationality in the Wikipedia article are not as useful as they are for professions. A typical example could be the term “German-American” to describe (primary) Americans with German roots. Not only is the secondary nationality mentioned first, the article may also discuss the person’s ancestry further, frequently mentioning that country. Statistical signals for the primary nationality, like

mentions of the place of residence or typical institutions are only considered by our more advanced approaches.

5.3 Refined analysis

We also conducted an analysis of the “clear cases”, that is, the triples, where the score from the crowdsourcing judges were either 6 or 7 or 1 or 0 (all judges, except at most one, agree). We analyzed the percentage of triples, which got the “reverse” scores for these triples, that is 0-1 for 6-7 and 6-7 for 0-1. The figures were very similar to those already captured by the Acc-4 figure in Table 4. Recall that a score deviation of more than 4 can only happen for the extreme scores. Thus, no new insights were provided by this analysis.

We also analyzed the dependency between accuracy and popularity of a person.

Method	Popularity Bucket					
	1	2	3	4	5	6
Words Counting	58%	76%	71%	71%	76%	81%
Count Combined	63%	81%	72%	68%	76%	81%
Words MLE	68%	84%	76%	79%	84%	76%
MLE Combined	74%	84%	75%	79%	84%	79%

Method	Popularity Bucket					
	7	8	9	10	11	12
Words Counting	77%	75%	70%	72%	82%	83%
Count Combined	86%	78%	76%	76%	78%	78%
Words MLE	76%	74%	69%	77%	79%	82%
MLE Combined	82%	75%	76%	80%	80%	82%

Table 7: Accuracy-2 for the *profession* relation, breakdown by person popularity. Popularity Bucket i contains persons with a number of occurrences between 2^{i+2} and 2^{i+3} . Buckets 1 and 12 are special cases and contain persons with less than 2^4 , resp. more than 2^{14} , occurrences.

The buckets of popularity used in Table 7 pertain to the buckets from our selection for the crowdsourcing task (see Section 3.1). The breakdown shows that approaches function well on all persons with more than 16 occurrences in the text. This corresponds to buckets 2 and above. While more text is always better for our statistical models, we observe that the minimum required to work reasonably well, is already very low. The persons in bucket 1 with very little associated text are harder to get correct. We can observe two things: *Words Counting* has more problems than *Words MLE*. Further, the combination with the classifier specifically helps to even out the problems with entities in this bucket.

5.4 Variants

Our main results show a gap of 11-14% between our best method, and what human judges can achieve. We have experimented with numerous variations of the methods described in Section 4 and 5.1. None of these variations turned out to give an improvement.

Stemming. All approaches from Section 4 can be used with and without stemming. We tried both the Porter [22] and the Lancaster stemmer [19]. For all methods, stemming (in either variant) did more harm than good.

Word pairs instead of just words. One significant source of errors in the methods from Section 4 is that *single* words are ambiguous indicators for certain triples. For example, the word *film* is a

strong indicator for both *Actor* and *Film Director*. Some persons have both professions. To distinguish between the two, the context of the word *film* is important. One obvious approach to address this is to consider pairs of words as features, instead of just single words. For the *profession* relation, the influence on both the score-based and the rank-based measure was insignificant though. For the *nationality* relation, results became significantly worse when using word pairs. This is understandable, since for the majority of nationalities single words are perfectly suited to uniquely identify them (e.g., *Canadian*, *Italian*, *German*).

TF versus TF-IDF In our evaluation, we used tf-idf for all our features. We also experimented with tf features, as well as with variations of tf-idf that give more emphasis to the idf part or that dampened the tf factor like in BM25. Results were either unaffected or became slightly worse.

Non-linear score mappings. In Section 3, we discussed two score mappings: Maplin and Maplog. The latter uses the mapping $x \mapsto \log_2 x$. The rationale was that the MLE approach produces raw scores (the probabilities) that grow super-linearly with the actual score. We experimented with a variety of other non-linear score mappings to capture this behavior, in particular, $x \mapsto x^\alpha$, for various value of α . The results were similar but never significantly better than for our Maplog.

Sentences vs. contexts In our methods in Section 4, we consider only those words that occur “in the same context” as the entity in question. We experimented with two realizations of this context: co-occurrence in the same *sentence* and co-occurrence in the same *context*, as described in Section 4. Contexts consistently outperformed sentences by a few percent. The improvement was more pronounced for popular entities, where we have many occurrences of the indicator words. This is in accordance with the theory from [4]. Co-occurrence in the same sentences does not necessarily mean that the two entities / words that co-occur have something to do with each other. When they co-occur in the same context, they have, by the definition of context from [4].

Whole corpus vs. only the Wikipedia article Almost all human judges indicated that they only used the Wikipedia article of a person to make their decision (95% of all decisions). For our methods from Section 4, we considered co-occurrences anywhere in the text corpus. When restricting to only the Wikipedia page of the respective person, like the human judges did, results consistently became much worse. This is easy to understand, since all our methods are statistics-based, and the whole corpus simply provides more information, and hence also more certainty to make a good decision.

Treating hierarchy / specialization In Freebase, a person’s professions may include generalizations of another profession that is listed. For example, for John Lennon there are (among other professions) singer-songwriter, keyboard player, guitarist and also musician. Like for Hierarchical Classification [18] problems, two obvious approaches are to (1) ignore the hierarchy and classify each triple and to (2) only classify specializations and infer a score for parent professions. We have experimented with both approaches and found that there is only little difference. If the same inference rules from (2) are used on both, our scores and the human judgments (human judgments are corrected towards a higher score if the human judges have given that higher score to a specialization of the profession), this increases the final scores by a few percent (e.g., 82% for *MLE Combined*). Otherwise, there is not much difference.

Knowledge-base facts An obvious alternative to using text for computing triple scores is to use information from the knowledge base. However, we have made the experience that 1) only few properties

are reflected by other facts in the knowledge base and 2) even in cases where they are, they are typically only available for popular instances. For example, an indication for the relevance of an actor may be in how many movies he acted, or how many awards he won for them. But there are no such facts in the knowledge base for a geologist, a surgeon, or a firefighter. Furthermore, in Freebase, out of 612K persons with more than one profession, 400K have less than 10 and 243K less than 6 other facts (besides type and profession information).

5.5 Manual Error Analysis

For all our methods from Section 4, as well as for all the variants discussed in Section 5.4, we conducted a manual error analysis. We discovered two main sources of errors.

Wrong reference An indicator word sometimes co-occurs with the person in question, but is actually relating to another person. For example, Michael Jackson is listed as a Film Director, which is certainly not one of his main professions. The Wikipedia article about him contains many mentions of the word *directed* or *director* though. But most of them relate not to him but to a person directing one of his shows.

Wrong meaning An indicator word is sometimes used with another meaning. For example, John F. Kennedy is listed as a Military Officer, which is certainly not one of his main professions. The Wikipedia article contains many mentions of variants of the word *military* though. But most of them relate to political actions during his presidency with respect to the military.

Some of the variants discussed in Section 5.4 mitigate the effect of these problems somewhat. For example, restricting co-occurrence to semantic contexts instead of full sentences helps the wrong-reference problem in several instances. Or, considering word pairs instead of single words helps with the wrong-meaning problem in several instances. However, a significant number of instances remain, where deep natural language understanding appears to be required. Consider the sentences “<Screenwriter X> was very happy with how his script was visualized” vs “<Film Director Y> was praised a lot for how he visualized the script”. It is hard to understand the roles of the persons in these sentences. But it is necessary to correctly decide if the sentence is evidence for a profession *Screenwriter* or *Film Producer*.

6. CONCLUSION

We have studied the problem of computing relevance scores for knowledge-base triples from type-like relations. We have shown that we can compute good such scores in an unsupervised manner from a text corpus with an accuracy of about 80%.

We have described various attempts to further improve the accuracy of our triple scores, towards the over 90% accuracy achieved by human judges. These attempts, together with a manual error analysis, suggest that deep natural language understanding is required to close this gap. In particular, we identified two main error sources: *wrong reference* and *wrong meaning* of indicator words. We consider these a hard but worthwhile task for future work.

The focus of this work is on scores for individual triples. This problem is practically relevant on its own and also hard enough on its own. Still, it would be interesting to follow up on previous work about integrating such triple scores into a meaningful ranking for complex knowledge-base queries.

7. REFERENCES

- [1] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.
- [2] K. Balog, P. Serdyukov, and A. P. de Vries. Overview of the TREC 2011 Entity Track. In *TREC*, 2011.
- [3] H. Bast, F. Bärle, B. Buchhold, and E. Haussmann. Broccoli: Semantic full-text search at your fingertips. *CoRR*, abs/1207.2615, 2012.
- [4] H. Bast and E. Haussmann. Open information extraction via contextual sentence decomposition. In *ICSC*, pages 154–159, 2013.
- [5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.
- [6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. In *NIPS*, pages 601–608, 2001.
- [7] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.
- [8] J. P. Cedeño and K. S. Candan. R²DF framework for ranked path queries over weighted RDF graphs. In *WIMS*, page 40, 2011.
- [9] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.
- [10] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. R. Stat. Soc.*, pages 1–38, 1977.
- [11] R. Q. Dividino, G. Gröner, S. Scheglmann, and M. Thimm. Ranking RDF with provenance via preference aggregation. In *EKAW*, pages 154–163, 2012.
- [12] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 7(10):881–892, 2014.
- [13] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum. Language-model-based ranking for queries on RDF-graphs. In *CIKM*, pages 977–986, 2009.
- [14] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee. Comparing and aggregating rankings with ties. In *PODS*, pages 47–58, 2004.
- [15] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.
- [16] T. Franz, A. Schultz, S. Sizov, and S. Staab. TripleRank: Ranking semantic web data by tensor decomposition. In *ISWC*, pages 213–228, 2009.
- [17] T. Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57, 1999.
- [18] C. N. S. Jr. and A. A. Freitas. A survey of hierarchical classification across different application domains. *Data Min. Knowl. Discov.*, 22(1-2):31–72, 2011.
- [19] C. D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.
- [20] S. Parsons. Current approaches to handling imperfect information in data and knowledge bases. *IEEE Trans. Knowl. Data Eng.*, 8(3):353–372, 1996.
- [21] D. Ramage, D. L. W. Hall, R. Nallapati, and C. D. Manning. Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP*, pages 248–256, 2009.
- [22] C. J. Van Rijsbergen, S. E. Robertson, and M. F. Porter. *New models in probabilistic information retrieval*. Computer Laboratory, University of Cambridge, 1980.

Semantic Search on Text and Knowledge Bases

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

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

Elmar Haussmann
University of Freiburg
haussmann@cs.uni-freiburg.de

Contents

1	Introduction	120
1.1	Motivation for this Survey	120
1.2	Scope of this Survey	123
1.3	Overview of this Survey	126
1.4	Glossary	128
2	Classification by Data Type and Search Paradigm	131
2.1	Data Types and Common Datasets	132
2.2	Search Paradigms	143
2.3	Other Aspects	148
3	Basic NLP Tasks in Semantic Search	150
3.1	Part-of-Speech Tagging and Chunking	151
3.2	Named-Entity Recognition and Disambiguation	153
3.3	Sentence Parsing	158
3.4	Word Vectors	162
4	Approaches and Systems for Semantic Search	167
4.1	Keyword Search in Text	168
4.2	Structured Search in Knowledge Bases	173
4.3	Structured Data Extraction from Text	179
4.4	Keyword Search on Knowledge Bases	190

4.5	Keyword Search on Combined Data	196
4.6	Semi-Structured Search on Combined Data	203
4.7	Question Answering on Text	207
4.8	Question Answering on Knowledge Bases	212
4.9	Question Answering on Combined Data	220
5	Advanced Techniques used for Semantic Search	227
5.1	Ranking	227
5.2	Indexing	234
5.3	Ontology Matching and Merging	239
5.4	Inference	243
6	The Future of Semantic Search	247
6.1	The Present	247
6.2	The Near Future	249
6.3	The Not So Near Future	250
	Acknowledgements	253
	Appendices	254
A	Datasets	254
B	Standards	256
	References	257

Abstract

This article provides a comprehensive overview of the broad area of semantic search on text and knowledge bases. In a nutshell, semantic search is “search with meaning”. This “meaning” can refer to various parts of the search process: understanding the query (instead of just finding matches of its components in the data), understanding the data (instead of just searching it for such matches), or representing knowledge in a way suitable for meaningful retrieval.

Semantic search is studied in a variety of different communities with a variety of different views of the problem. In this survey, we classify this work according to two dimensions: the type of data (text, knowledge bases, combinations of these) and the kind of search (keyword, structured, natural language). We consider all nine combinations. The focus is on fundamental techniques, concrete systems, and benchmarks. The survey also considers advanced issues: ranking, indexing, ontology matching and merging, and inference. It also provides a succinct overview of natural language processing techniques that are useful for semantic search: POS tagging, named-entity recognition and disambiguation, sentence parsing, and word vectors.

The survey is as self-contained as possible, and should thus also serve as a good tutorial for newcomers to this fascinating and highly topical field.

H. Bast, B. Buchhold, E. Haussmann. *Semantic Search on Text and Knowledge Bases*. Foundations and Trends[®] in Information Retrieval, vol. 10, no. 2-3, pp. 119–271, 2016.

DOI: 10.1561/15000000032.

1

Introduction

1.1 Motivation for this Survey

This is a survey about the broad field of semantic search. Semantics is the study of meaning.¹ In a nutshell, therefore, it could be said that semantic search is *search with meaning*.

Let us first understand this by looking at the opposite. Only a decade ago, search engines, including the big web search engines, were still mostly *lexical*. By lexical, we here mean that the search engine looks for literal matches of the query words typed by the user or variants of them, without making an effort to understand what the whole query actually means.

Consider the query *university freiburg* issued to a web search engine. Clearly, the homepage of the University of Freiburg is a good match for this query. To identify this page as a match, the search engine does not need to understand what the two query words *university* and *freiburg* actually mean, nor what they mean together. In fact, the university homepage contains these two words in its title (and, as a

¹The word comes from the ancient greek word *sēmantikós*, which means *important*.

matter of fact, no other except the frequent word *of*). Further, the page is at the top level of its domain, as can be seen from its URL: <http://www.uni-freiburg.de>. Even more, the URL consists of parts of the query words. All these criteria are easy to check, and they alone make this page a very good candidate for the top hit of this query. No deeper understanding of what the query actually “meant” or what the homepage is actually “about” were needed.²

Modern search engines go more and more in the direction of accepting a broader variety of queries, actually trying to “understand” them, and providing the most appropriate answer in the most appropriate form, instead of just a list of (excerpts from) matching documents.

For example, consider the two queries *computer scientists* and *female computer scientists working on semantic search*. The first query is short and simple, the second query is longer and more complex. Both are good examples of what we would call semantic search. The following discussion is independent of the exact form of these queries. They could be formulated as keyword queries like above. They could be formulated in the form of complete natural language queries. Or they could be formulated in an abstract query language. The point here is what the queries are asking for.

To a human, the intention of both of these queries is quite clear: the user is (most likely) looking for scientists of a certain kind. Probably a list of them would be nice, with some basic information on each (for instance, a picture and a link to their homepage). For the query *computer scientists*, Wikipedia happens to provide a page with a corresponding list and matching query words.³ Correspondingly, the list is also contained in DBpedia, a database containing the structured knowledge from Wikipedia. But in both cases it is a manually compiled list, limited to relatively few better-known computer scientists. For the second query (*female computer scientists working on semantic search*), there is no single web page or other document with a corresponding

²In this simple example, we are leaving aside the important issue of *spam*. That is, someone deliberately putting misleading keywords in the title or even in the URL, in order to fool search engines, and thus users, to consider the web page relevant. Note that this query could also be solved using clickthrough data; see Section 1.2.2.

³http://en.wikipedia.org/wiki/List_of_computer_scientists

list, let alone one matching the query words. Given the specificity of the query, it is also unlikely that someone will ever manually compile such a list (in whatever format) and maintain it. Note that both lists are constantly changing over time, since new researchers may join any time.

In fact, even individual web pages matching the query are unlikely to contain most of the query words. A computer scientist does not typically put the words *computer scientist* on his or her homepage. A female computer scientist is unlikely to put the word *female* on her homepage. The homepage probably has a section on that particular scientist's research interests, but this section does not necessarily contain the word *working* (maybe it contains a similar word, or maybe no such word at all, but just a list of topics). The topic *semantic search* will probably be stated on a matching web page, though possibly in a different formulation, for example, *intelligent search* or *knowledge retrieval*.

Both queries are thus good examples, where search needs to go beyond mere lexical matching of query words in order to provide a satisfactory result to the user. Also, both queries (in particular, the second one) require that information from several different sources is brought together to answer the query satisfactorily. Those information sources might be of different kinds: (unstructured) text as well as (structured) knowledge bases.

There is no exact definition of what semantic search is. In fact, semantic search means a lot of different things to different people. And researchers from many different communities are working on a large variety of problems related to semantic search, often without being aware of related work in other communities. *This is the main motivation behind this survey.*

When writing the survey, we had two audiences in mind: (i) newcomers to the field, and (ii) researchers already working on semantic search. Both audiences should get a comprehensive overview of which approaches are currently pursued in which communities, and what the current state of the art is. Both audiences should get pointers for further reading wherever the scope of this survey (defined in Section 1.2

right next) ends. But we also provide explanations of the underlying concepts and technologies that are necessary to understand the various approaches. Thus, this survey should also make a good tutorial for a researcher previously unfamiliar with semantic search.

1.2 Scope of this Survey

1.2.1 Kinds of Data

This survey focuses on semantic search on text (in natural language) or knowledge bases (consisting of structured records). The two may also be combined. For example, a natural language text may be enriched with semantic markup that identifies mentions of entities from a knowledge base. Or several knowledge bases with different schemata may be combined, like in the Semantic Web. The types of data considered in this survey are explained in detail in Section 2.1 on *Data Types and Common Datasets*.

This survey does *not* cover search on images, audio, video, and other objects that have an inherently non-textual representation. This is not to say that semantic search is not relevant for this kind of data; quite the opposite is true. For example, consider a user looking for a picture of a particular person. Almost surely, the user is not interested in the precise arrangements of pixels that are used to represent the picture. She might not even be interested in the particular angle, selection, or lighting conditions of the picture, but only in the object shown. This is very much “semantic search”, but on a different kind of data. There is some overlap with search in textual data, including attempts to map non-textual to textual features and the use of text that accompanies the non-textual object (e.g., the caption of an image). But mostly, search in non-textual data is a different world that requires quite different techniques and tools.

A special case of image and audio data are scans of text documents and speech. The underlying data is also textual⁴ and can be extracted using optical character recognition (OCR) and automatic speech recognition (ASR) techniques. We do not consider these techniques in this

⁴Leaving aside aspects like a particular writing style or emotions when talking.

survey. However, we acknowledge that “semantic techniques”, as described in this survey, can be helpful in the text recognition process. For example, in both OCR and ASR, a semantic understanding of the possible textual interpretations can help to decide which interpretation is the most appropriate.

1.2.2 Kinds of Search

There are three types of queries prevailing in semantic search: keyword, structured, and natural language. We cover the whole spectrum in this survey; see Section 2.2 on *Search Paradigms*.

Concerning the kind of results returned, we take a narrower view: we focus on techniques and systems that are *extractive* in the sense that they return elements or excerpts from the original data. Think of the result screen from a typical web search engine. The results are nicely arranged and partly reformatted, so that we can digest them properly. But it’s all excerpts and elements from the web pages and knowledge bases being searched in the background.

We only barely touch upon the analysis of query logs (queries asked) and clickthrough data (results clicked). Such data can be used to derive information on what users found relevant for a particular query. Modern web search engines leverage such information to a significant extent. This topic is out of scope for this survey, since an explicit “understanding” of the query or the data is not necessary. We refer the user to the seminal paper of Joachims [2002] and the recent survey of Silvestri [2010].

There is also a large body of research that involves the complex synthesis of new information, in particular, text. For example, in *automatic summarization*, the goal is to summarize a given (long) text document, preserving the main content and a consistent style. In *multi-document summarization*, this task is extended to multiple documents on a particular topic or question. For example, *compile a report on drug trafficking in the united states over the past decade*. Apart from collecting the various bits and pieces of text and knowledge required to answer these questions, the main challenge becomes to compile these into a compact and coherent text that is well comprehensible for hu-

mans. Such non-trivial automatic content synthesis is out of scope for this survey.

1.2.3 Further inclusion criteria

As just explained, we focus on semantic search on text and knowledge bases that retrieves elements and excerpts from the original data. But even there we cannot possibly cover all existing research in depth.

Our inclusion criteria for this survey are very practically oriented, with a focus on fundamental techniques, datasets, benchmarks, and systems. Systems were selected with a strong preference for those evaluated on one of the prevailing benchmarks or that come with a working software or demo. We provide quantitative information (on the benchmarks and the performance and effectiveness of the various systems) wherever possible.

We omit most of the history and mostly focus on the state of the art. The historical perspective is interesting and worthwhile in its own right, but the survey is already long and worthwhile without this. However, we usually mention the first system of a particular kind. Also, for each of our nine categories (explained right next, in Section 1.3), we describe systems in chronological order and make sure to clarify the improvements of the newer systems over the older ones.

1.2.4 Further Reading

The survey provides pointers for further reading at many places. Additionally, we provide here a list of well-known conferences and journals, grouped by research community, which are generally good sources for published research on the topic of this survey and beyond. In particular, the bibliography of this survey contains (many) references from each of these venues. This list is by no means complete, and there are many good papers that are right on topic but published in other venues.

Information Retrieval: SIGIR, CIKM, TREC, TAC, FNTIR.

Web and Semantic Web: WWW, ISWC, ESWC, AAI, JWS.

Computer linguistics: ACL, EMNLP, HLT-NAACL.

Databases / Data Mining: VLDB, KDD, SIGMOD, TKDE.

1.3 Overview of this Survey

Section 1.4 provides a *Glossary* of terms that are strongly related to semantic search. For each of these, we provide a brief description together with a pointer to the relevant passages in the survey. This is useful for readers who specifically look for material on a particular problem or aspect.

Section 2 on *Classification by Data Type and Search Paradigm* describes the two main dimensions that we use for categorizing research on semantic search:

Data type: text, knowledge bases, and combined data.

Search paradigm: keyword, structured, and natural language search.

For each data type, we provide a brief characterization and a list of frequently used datasets. For each search paradigm, we provide a brief characterization and one or two examples.

Section 3 on *Basic NLP Tasks in Semantic Search* gives an overview of: part-of-speech (POS) tagging, named-entity recognition and disambiguation (NER+NED), parsing the grammatical structure of sentences, and word vectors / embeddings. These are used as basic building blocks by various (though not all) of the approaches described in our main Section 4. We give a brief tutorial on each of these tasks, as well as a succinct summary of the state of the art.

Section 4 on *Approaches and Systems for Semantic Search* is the core section of this survey. We group the many approaches and systems that exist in the literature by data type (three categories, see above) and search paradigm (three categories, see above). The resulting nine combinations are shown in Figure 1.1. In a sense, this figure is the main signpost for this survey. Note that we use *Natural Language Search* and *Question Answering* synonymously in this survey. All nine subsections share the same sub-structure:

Profile ... a short characterization of this line of research

Techniques ... what are the basic techniques used

Systems ... a concise description of milestone systems or software

Benchmarks ... existing benchmarks and the best results on them

	Keyword Search	Structured Search	Natural Lang. Search
Text	Section 4.1 Keyword Search on Text	Section 4.3 Structured Data Extraction from Text	Section 4.7 Question Answering on Text
Knowledge Bases	Section 4.4 Keyword Search on Knowledge Bases	Section 4.2 Structured Search on Knowledge Bases	Section 4.8 Question Answering on Knowledge Bases
Combined Data	Section 4.5 Keyword Search on Combined Data	Section 4.6 Semi-Struct. Search on Combined Data	Section 4.9 Question Answering on Combined Data

Figure 1.1: Our basic classification of research on semantic search by underlying data (rows) and search paradigm (columns). The three data types are explained in Section 2.1, the three search paradigms are explained in Section 2.2. Each of the nine groups is discussed in the indicated subsection of our main Section 4.

Section 5 on *Advanced Techniques for Semantic Search* deals with: *ranking* (in semantic entity search), *indexing* (getting not only good results but getting them fast), *ontology matching and merging* (dealing with multiple knowledge bases), and *inference* (information that is not directly contained in the data but can be inferred from it). They provide a deeper understanding of the aspects that are critical for results of high quality and/or with high performance.

Section 6 on *The Future of Semantic Search* provides a very brief summary of the state of the art in semantic search, as described in the main sections of this survey, and then dares to take a look into the near and the not so near future.

The article closes with a long list of 218 references. Datasets and standards are not listed as part of the References but separately in the Appendices. In the PDF of this article, all citations in the text are clickable (leading to the respective entry in the References), and so are

most of the titles in the References (leading to the respective article on the Web). In most PDF readers, *Alt+Left* brings you back to the place of the citation.

The reader may wonder about possible reading orders and which sections depend upon which. In fact, each of the six sections of this survey is relatively self-contained and readable on its own. This is true even for each of the nine subsections (one for each kind of semantic search, according to our basic classification) of the main Section 4. However, when reading such a subsection individually, it is a good idea to prepend a quick read of those subsections from Section 2 that deal with the respective data type and search paradigm: they are short and easy to read, with instructive examples. Readers looking for specific information may find the glossary, which comes right next, useful.

1.4 Glossary

This glossary provides a list of techniques or aspects that are strongly related to semantic search but non-trivial to find using our basic classification. For each item, we provide a very short description and a pointer to the relevant section(s) of the survey.

Deep learning for NLP: natural language processing using (deep) neural networks; used for the word vectors in Section 3.4; some of the systems in Section 4.8 on *Question Answering on Knowledge Bases* use deep learning or word vectors; apart from that, deep NLP is still used very little in actual systems for semantic search, but see Section 6 on *The Future of Semantic Search*.

Distant supervision: technique to derive labeled training data using heuristics in order to learn a (supervised) classifier; the basic principle and significance for semantic search is explained in Section 4.3.2 on *Systems for Relationship Extraction from Text*.

Entity resolution: identify that two different strings refer to the same entity; this is used in Section 4.3.4 on *Knowledge Base Construction* and discussed more generally in Section 5.4 on *Ontology Matching and Merging*.

Entity search/retrieval: search on text or combined data that aims at a particular entity or list of entities as opposed to a list of documents; this applies to almost all the systems in Section 4 that work with combined data or natural language queries⁵; see also Section 5.1, which is all about ranking techniques for entity search.

Knowledge base construction: constructing or enriching a knowledge base from a given text corpus; basic techniques are explained in Section 4.3.1; systems are described in Section 4.3.4.

Learning to rank for semantic search: supervised learning of good ranking functions; several applications in the context of semantic search are described in Section 5.1.

Ontology merging and matching: reconciling and aligning naming schemes and contents of different knowledge bases; this is the topic of Section 5.3.

Paraphrasing or synonyms: identifying whether two words, phrases or sentences are synonymous; systems in Section 4.8 on *Question Answering on Knowledge Bases* make use of this; three datasets that are used by systems described in this survey are: Patty [2013] (paraphrases extracted in an unsupervised fashion), Paralex [2013] (question paraphrases), and CrossWikis [2012] (Wikipedia entity anchors in multiple languages).

Question answering: synonymous with natural language search in this survey; see Section 2.2.3 for a definition; see Sections 4.7, 4.8, and 4.9 for research on question answering on each of our three data types.

Reasoning/Inference: using reasoning to infer new triples from a given knowledge base; this is the topic of Section 5.4.

Semantic parsing: finding the logical structure of a natural language query; this is described in Sections 4.8 on *Question Answering on Knowledge Bases* and used by many of the systems there.

Semantic web: a framework for explicit semantic data on the web; this kind of data is described in Section 2.1.3; the systems described

⁵A search on a knowledge base naturally returns a list of entities, too. However, the name *entity search* is usually only used when (also) text is involved and returning lists of entities is not the only option.

in Section 4.5 deal with this kind of data; it is important to note that many papers / systems that claim to be about semantic web data are actually dealing only with a single knowledge base (like DBpedia, see Table 2.2), and are hence described in the sections dealing with search on knowledge bases.

Information extraction: extracting structured information from text; this is exactly what Section 4.3 on *Structured Data Extraction from Text* is about.

XML retrieval: search in nested semi-structured data (text with tag pairs, which can be arbitrarily nested); the relevance for semantic search is discussed in Section 4.5.3 in the context of the INEX series of benchmarks.

2

Classification by Data Type and Search Paradigm

In this section, we elaborate on our basic classification of semantic search research and systems. The classification is along two dimensions:

Data type: text, knowledge bases, or combined data

Search paradigm: keyword, structured, and natural language search

In Section 2.1, we explain each of the three data types, providing a list of frequently used datasets for each type. In Section 2.2, we explain each of the three search paradigms along with various examples. The resulting nine combinations are shown in Figure 1.1.

Why this Classification

Coming up with this simple classification was actually one of the hardest tasks when writing this survey. Our goal was to group together research that, from a technical perspective, addresses similar problems, with a relatively clear delineation between different groups (much like in *clustering* problems). Most of the systems we looked at clearly fall into one of our categories, and no other classification we considered (in particular, refinements of the one from Figure 1.1) had that property. Of course, certain “gray zones” between the classes are inevitable;

these are discussed in the respective sections. For example, there is an interesting gray zone between *keyword* and *natural language* queries, which is discussed at the beginning of Section 2.2. Also, it is sometimes debatable whether a dataset is a single knowledge base or a combination of different knowledge bases, which counts as combined data in our classification; this is discussed in Section 2.1.2 on *Knowledge Bases*.

Also note that some other natural aspects are implicitly covered by our classification: for example, the *type of result* is largely implied by the type of data and the kind of search. Another complication (or rather, source of confusion) is terminology mixup. To give just one example, there is a huge body of research on the Semantic Web, but much of this work is actually concerned with a single knowledge base (like DBpedia, see Table 2.2), which requires mostly different techniques compared to true semantic web data, which is huge and extremely heterogeneous. Our Glossary in Section 1.4 should help to resolve such mixups, and, more generally, to locate (in this survey) material on a given technique or aspect.

Yet other aspects are orthogonal to our primary classification, for example: interactivity, faceted search, and details of the result presentation. These could be added with advantage to almost any system for semantic search. We briefly discuss such aspects in Section 2.3.

2.1 Data Types and Common Datasets

This section explains each of the three basic data types used in our classification above: natural language text, knowledge bases, and combinations of the two. For each type, we provide a list of frequently used datasets. All datasets are listed in a dedicated subsection of the References section. In the PDF of this article, the references in the tables below are clickable and lead to the corresponding entry in the Appendix.

2.1.1 Text

Definition 2.1. For the purpose of this survey, *text* is a collection of documents containing text, typically written in natural language. The

text need not be orthographically or grammatically correct. It may contain typical punctuation and light markup that exhibits some high-level structure of the text, like title and sections. There may also be hyperlinks between documents.

Remark: If there is markup that provides fine-grained annotations of parts of the text (e.g., linking an entity mention to a knowledge base), we count this as *Combined Data*, as discussed in Section 2.1.3.

Text is ubiquitous in the cyberworld, because it is the natural form of communication between humans. Typical examples are: news articles, e-mails, blog posts, tweets, and all kinds of web pages.

Web pages pose several additional challenges, like boilerplate content (e.g., navigation, headers, footers, etc., which are not actual content and can be misleading if not removed), spam, and dynamically generated content. We do not discuss these aspects in this survey. On the positive side, the hyperlinks are useful for search in general. Techniques for exploiting hyperlinks in the context of semantic search are discussed in Section 5.1.3 on *Ranking of Interlinked Entities*.

Commonly Used Datasets

Table 2.2 lists some collections of text documents that are often used in research on semantic search.

Reference	Documents	Size	zip	Type
[AQUAINT, 2002]	1.0 million	3.0 GB	n	news articles
[AQUAINT2, 2008]	0.9 million	2.5 GB	n	news articles
[Blogs06, 2006]	3.2 million	25 GB	n	blog posts
[ClueWeb, 2009]	1.0 billion	5.0 TB	y	web pages
[ClueWeb, 2012]	0.7 billion	5.0 TB	y	web pages
[CommonCrawl, 2007]	2.6 billion	183 TB	n	web pages
[Stream Corpus, 2014]	1.2 billion	16.1 TB	y	web pages ¹

Table 2.1: Datasets of natural language text used in research on semantic search.

The two AQUAINT datasets were heavily used in the TREC benchmarks dealing with question answering on text; see Section 4.7. The ClueWeb datasets are (at the time of this writing) the most used web-scale text collections. The CommonCrawl project provides regular snapshots (at least yearly) of a large portion of the Web in various languages. The Stream Corpus has been used in the TREC Knowledge Base Acceleration tracks (see Section 4.3 on *Structured Data Extraction from Text*) where knowledge about entities can evolve over time.

2.1.2 Structured Data / Knowledge Bases

Definition 2.2. For the purpose of this survey, a *knowledge base* is a collection of records in a database, which typically refer to some kind of “knowledge” about the world. By convention, records are often stored as triples in the form *subject predicate object*.

To qualify as a knowledge base, identifiers should² be used consistently: that is, the same entity or relation should have the same name in different records. Collections of records / triples from different sources with different naming schemes are counted as *Combined Data*, which is discussed in Section 2.1.3.

Here are four example records from the Freebase dataset (see Table 2.2 below). The *ns:* is a common prefix, and the corresponding identifiers are URIs; see the subsection on data formats below.

<i>ns:m.05b6w</i>	<i>ns:type.object.name</i>	<i>"Neil Armstrong"</i>
<i>ns:m.0htp</i>	<i>ns:type.object.name</i>	<i>"Astronaut"</i>
<i>ns:m.05b6w</i>	<i>ns:people.person.profession</i>	<i>ns:m.0htp</i>
<i>ns:m.05b6w</i>	<i>ns:people.person.date_of_birth</i>	<i>"08-05-1930"</i>

Note that by the consistent use of identifiers we can easily derive information like *a list of all astronauts* or *astronauts born before a certain date*. We briefly discuss some related terminology and finer points.

¹Web pages are timestamped, which allows treating the corpus as a stream of documents.

²A small fraction of inconsistencies are unavoidable in a large knowledge base and hence acceptable.

Ontologies: an ontology is the (typically hierarchical) system of types and relations behind a knowledge base. For example, the fact that astronauts are persons and that all persons are entities are typical ontological statements. WordNet [Miller, 1992] is a large ontology of the concepts of general-purpose real-world knowledge. In a sense, an ontology is therefore also a knowledge base, but on more “abstract” entities. The distinction is not always sharp, however. For example, WordNet also contains statements about “concrete entities”, like in a typical knowledge base. Throughout this survey, we will consistently use the term knowledge base when referring to collections of records as defined above.

n -ary relations: It is easy to see that one can break down any structured data into triples, without loss of information. This is an instance of what is called *reification*. An example is given at the end of Section 2.1.3 (Christoph Waltz’s Oscar).

n -tuples with $n > 3$: some knowledge bases also store tuples with more than three components. Typical uses are: adding provenance information (the data source of a triple), adding spatial or temporal information, assigning a unique id to a triple.

Triples vs. facts. vs. statements: the triples or n -tuples are sometimes referred to (somewhat optimistically) as facts or (more carefully) as statements. This does not mean that they are necessarily true. They may have entered the knowledge base by mistake, or they may just express an opinion. Still, very often they are “facts” in the common sense and it usually makes sense to think of them like that.

Graph representation: a knowledge base can also be thought of as a graph, where the nodes are the entities and the edges are the relations. When n -ary relations are involved, with $n > 2$, these edges become hyperedges (connecting more than two entities) and the graph becomes a hypergraph.

Commonly Used Datasets

Table 2.2 lists some often used knowledge bases. It is sometimes debatable when a dataset is a single knowledge base (as discussed in this

section) or a combination of different knowledge bases (as discussed in the next section). Our criterion, according to Definition 2.2 above, is whether the bulk of the data follows a consistent ontology / naming scheme. For example, the bulk of DBpedias’s knowledge is stored in *dbpedia-owl:...* relations which are used consistently across entities. But there are also numerous *dbprop:...* relations, which correspond to a wide variety of properties from the Wikipedia infoboxes, which do not follow a strict underlying schema. Similarly, Freebase has numerous relations from its “*base*” domain, which partly fill in some interesting gaps and partly provide redundant or even confusing information.³

Reference	#Triples	#Entities	Size	Type
[YAGO, 2007]	20 M	2.0 M	1.4 GB	Wikipedia
[YAGO2s, 2011]	447 M	9.8 M	2.2 GB	Wikipedia
[DBpedia, 2007] ⁴	580 M	4.6 M	3.8 GB	Wikipedia
[GeoNames, 2006]	150 M	10.1 M	465 MB	geodata
[MusicBrainz, 2003]	239 M	45.0 M	4.1 GB	music
[UniProt, 2003]	19.0 B	3.8 B	130 GB	proteins
[Freebase, 2007]	3.1 B	58.1 M	30 GB	general
[Wikidata, 2012]	81 M	19.3 M	5.9 GB	general

Table 2.2: Knowledge bases used in research on semantic search. All sizes are of the compressed dataset.

We also remark that usually only a fraction of the triples in these knowledge bases convey “knowledge” in the usual sense. For example, the YAGO dataset contains about 3 million facts stating the length of each Wikipedia page. Freebase contains 10 million facts stating the keys of all Wikipedia articles. DBpedia has millions of *rdf:type* triples relating entities to the countless synsets from WordNet. Also, many facts are redundant. For example, in Freebase many relations have an

³For example, the type *base.surprisingheights.surprisingly_short_people* with only fifteen entities, including Al Pacino.

⁴The number of triples and entities are for the English version. The multilingual version features 3 billion triples and 38.8 million entities.

inverse relation with the same statements with subject and object reversed. According to Bordes and Gabrilovich [2015], the number of non-redundant triples in Freebase is 637 million, which is about one third of the total number stated in the table above.

On December 16, 2014 Google announced that it plans to merge the Freebase data into Wikidata and then stop accumulating new data in Freebase. Freebase became read-only on March 30, 2015. At the time of this writing, Wikidata was still relatively small, however.

Data Formats

A knowledge base can be stored in a general-purpose relational database management system (RDBMS), or in special-purpose so-called triple stores. The efficiency of the various approaches is discussed in Section 4.2 on *Structured Search on Knowledge Bases*.

On the Web, knowledge base data is often provided in the form of RDF (Resource Description Framework). RDF is complex, and we refer the reader to Wikipedia or W3C for a complete description. What is notable for this survey is that in RDF, identifiers are provided by a URI (Universal Resource Identifier) and hence globally unambiguous.

Also note that RDF is an abstract description model and language, not an explicit format. For practical purposes, many text serializations exist. The first such serialization was proposed in 1999 by the W3C and was based on XML, and thus very verbose. At the time of this writing, less verbose text serializations are commonly used:

N-triples: the triples are stored in a text file, with a space between subject, predicate, and object, and a simple dot to separate triples (usually one triple per line).

N-quads: like N-triples, but with one additional field per triple that provides an arbitrary context value (e.g., the source of the triple).

TSV: one triple or quad per line, with the three or four components separated by a tab. TSV is an abbreviation for tab-separated values.

Turtle: allows an explicit nested representation. Depending on the data, this can be more convenient for reading and producing than mere

triples or quads. The price is a more complex format that requires more complex parsers.

2.1.3 Combined Data

Text and knowledge bases are both natural forms to represent knowledge. Text is the most natural form for humans to produce information. A knowledge base is the most natural form to store information that is inherently structured in the first place. Therefore, it makes sense to combine data of these two types into a maximally comprehensive dataset. It also makes sense to consider multiple knowledge bases, since a single knowledge base is usually limited to a certain scope. Of course, it also makes sense to combine different text collections, but that is trivial since there is no common structure or naming scheme to obey.

Definition 2.3. For the purpose of this survey, *combined data* is obtained by one or both of the following two principles:

link: link a text to a knowledge base by recognizing mentions of entities from the knowledge base in the text and linking to them

mult: combine multiple knowledge bases with different naming schemes (such that the same entity or relation may exist with different names)

Both of these are used extensively in research in order to obtain what we call *combined data* here. In the list of commonly used datasets in Table 2.3 below, it is indicated which dataset makes use of which subset of these principles. Note that realizing “link” is equivalent to solving the named-entity recognition and disambiguation problem discussed in Section 3.2.

Commonly Used Datasets

Table 2.3 lists a number of popular datasets of the “combined” type. The number of “triples” for the Wikipedia LOD dataset, the two ClueWeb FACC datasets, and the FAKBA1 dataset is the number of entity mentions in the text that were linked to an entity from the knowledge base (YAGO and DBpedia for the Wikipedia LOD dataset, Freebase for the FACC and FAKBA1 dataset). Note that the ClueWeb

Reference	#Triples	Size	Type
[Wikipedia LOD, 2012]	70 million	61 GB	link
[ClueWeb09 FACC, 2013]	5.1 billion	72 GB	link
[ClueWeb12 FACC, 2013]	6.1 billion	92 GB	link
[FAKBA1, 2015]	9.4 billion	196 GB	link
[BTC, 2009]	1.1 billion	17 GB	mult
[BTC, 2010]	3.2 billion	27 GB	mult
[BTC, 2012]	1.4 billion	17 GB	mult
[WDC, 2012]	17.2 billion	332 GB	link+mult

Table 2.3: Commonly used datasets of the “combined” type. The last column indicates which combination principles were used, according to the typology explained at the beginning of the section.

FACC and FAKBA1 datasets only consist of the annotations and do not include the full text from ClueWeb or the Stream Corpus from the TREC Knowledge Base Acceleration track. The three BTC datasets were obtained from a crawl of the Semantic Web, started from a selection of seed URIs. Note that the BTC 2012 dataset contains all of DBpedia and a selection of Freebase, which are both listed in Table 2.2 as individual knowledge bases. The WDC (Web Data Commons) dataset is obtained by extracting structured data from CommonCrawl (see Table 2.1). Both BTC and WDC are considered “semantic web data”, which is explained in more detail below.

Text Linked to a Knowledge Base

The natural format to encode *link* information in text is XML. Here is an example excerpt from the Wikipedia LOD collection from Table 2.3 above.

```
<paragraph> Mt. Morris is home of the <link>
<wikilink href="13135902.xml">Illinois Freedom Bell</wikilink>
<dbpedia href="http://dbpedia.org/.../Illinois_Freedom_Bell">
</dbpedia><yago ref="Illinois_Freedom_Bell"></yago>
</link>, which is located in the town square. [...]</paragraph>
```

The main tasks of the INEX (Initiative for the Evaluation of XML retrieval) series of benchmarks, discussed in Section 4.5.3, work with this collection.

However, note that for the purposes of annotation, XML is a mere convention, not a necessity. For example, the two FACC collections from Table 2.3 above provide links between the ClueWeb collections (from Table 2.1) and Freebase (from Table 2.2) as follows:

<i>PDF</i>	<i>21089</i>	<i>21092</i>	<i>0.9976</i>	<i>m.0600q</i>
<i>FDA</i>	<i>21303</i>	<i>21306</i>	<i>0.9998</i>	<i>m.032mx</i>
<i>Food and Drug Administration</i>	<i>21312</i>	<i>21340</i>	<i>0.9998</i>	<i>m.032mx</i>

The first column is the name of the entity in the text, the second and third columns specify the byte offsets in the file, the fourth column is the confidence of the link, and the fifth column is the Freebase id.

Semantic Web

The Semantic Web (SW) is an effort to provide “combined data” in the sense above at a very large scale. The data from the Semantic Web is often also called *linked open data (LOD)*, because contents can be contributed and interlinked by anyone, just like web pages (but in a different format, see below). With respect to search, these are secondary aspects. Throughout this survey, we therefore relate to this kind of data as simply *semantic web data*. It makes uses of both principles of combining data, as defined above:

link: provided by semantic markup for ordinary web pages.

mult: anyone can contribute + absence of a global schema.

The “mult” principle is realized via RDF documents that can link to each other, just like ordinary web pages can link to each other. For example, here is an excerpt from the RDF page for the French city Embrun (the showcase page of the GeoNames knowledge base from Table 2.2). Note the use of prefixes like *rdf*: and *gn*: in the URI to keep the content compact and readable also for humans.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```

      xmlns:gn="http://www.geonames.org/ontology#" ... >
<gn:Feature rdf:about="http://sws.geonames.org/3020251/">
<gn:name>Embrun</gn:name>
<gn:countryCode>FR</gn:countryCode>
<gn:population>7069</gn:population>

```

The “link” principle is realized via semantic markup that is embedded into regular web pages. For example, here is an excerpt from an HTML page using so-called *Microdata* markup.

```

<title>Michael Slackenerny's Homepage</title>
<section itemscope itemtype="http://schema.org/Person">
Hi, my name is <span itemprop="name">Michael Slackenerny</span>.
I am a <span itemprop="jobTitle">postdoctoral student</span> at
<span itemprop="affiliation">Stanford Univ</span>.</section>

```

The four most widely used formats for semantic markup are as follows. The first three simply use HTML tags with existing or new attributes.

Microdata: uses the dedicated attributes *itemscope* and *itemprop*

Microformats: (ab)uses the *class* attribute

RDFa: uses RDF-style attributes *about*, *property*, and *contents*

JSON-LD: uses JavaScript to provide Microdata-like markup

The advantage of JSON-LD over the other formats is that it allows a cleaner separation of the ordinary content from the semantic content (much in the spirit of frameworks like CSS, or libraries like jQuery). As of this writing, all four of these formats are still widely used, and no clear winner has emerged yet.

The use of semantic markup has increased steadily over the last years. According to [Meusel, Petrovski, and Bizer, 2014], the number of RDF quads (all of the above, except JSON-LD) in the WDC dataset (see Table 2.3 below) has increased from 5.2 billion in the 2010 dataset, to 7.4 billion in 2012, to 17.2 billion in 2013. Between 2012 and 2013, the fraction of analyzed HTML pages and domains that use semantic markup has more than doubled, up to 26% of pages and 14% of domains in 2013. More recent statistics can be found in [Guha, Brickley, and

MacBeth, 2015]. Here, a sample of 10 billion pages from a combination of the Google index and Web Data Commons is analyzed. 31% of pages have Schema.org markup, as discussed in the next subsection.

Challenges of Semantic Web Data

It is important to understand that semantic web data is *not* a single knowledge base as defined in the previous section, because there is no consistent naming scheme. This is not by accident, but rather by design. Given the heterogeneity of contents and its providers on the Web, it seems illusory to establish standard names for everything and expect everyone to stick with it. The Semantic Web takes a minimalist approach in this respect: content providers can use whatever names they like, they only have to be globally unambiguous (URIs). We briefly discuss three resulting challenges.

Standards: One (mostly social) approach is to enable and encourage contributors to reuse existing schemes as much as possible. For common concepts, this is already happening. One of the earliest schemes was FOAF [2000] (Friend Of A Friend), which provided standard names for relations related to a person, like: given names, family name, age, and who knows who. A more recent and larger effort is Schema.org [2011], collaboratively launched by Bing, Google, and Yahoo. Schema.org provides URIs for concepts such as creative works, events, organizations, persons, places, product, reviews, which are relevant for many popular web search queries.

Explicit links: Another approach is to enable contributors to provide explicit links between different names for the same entities or concepts, via meta statements like `<owl:sameAs>`. An example for different names for the same entity is: *Barack Obama* and *Barack H. Obama* and *B. Obama*. Such links could also be identified by automatic methods. This is discussed in more detail in Section 5.3 on *Ontology Matching and Merging*.

Model mismatch: A more complex problem are relations, which can not only be named differently, but also modeled in different ways. For example, consider the YAGO and Freebase knowledge bases from Table

2.2. In YAGO, the relation *won award* is used to express who won which award. For example

Christoph Waltz won-award Oscar for Best Supporting Actor

In Freebase, that information is modeled via a so-called mediator object, which has id *ns:m.0r4b38v* in the following example. For the sake of readability, we replaced the ids of the other objects with human-readable names and shortened the relation names.

<i>Christoph Waltz</i>	<i>award</i>	<i>ns:m.0r4b38v</i>
<i>ns:m.0r4b38v</i>	<i>name</i>	<i>Oscar for Best Supporting Actor</i>
<i>ns:m.0r4b38v</i>	<i>year</i>	<i>2012</i>
<i>ns:m.0r4b38v</i>	<i>movie</i>	<i>Django Unchained</i>
<i>ns:m.0r4b38v</i>	<i>role</i>	<i>Dr. King Schulz</i>

The approach taken by YAGO is simpler, the approach taken by Freebase allows to capture more information.

2.2 Search Paradigms

We distinguish between three major search paradigms: keyword search, structured search, and natural language search. Each of them is explained in one of the following subsections. As a first approximation, think of the name of the paradigm to describe the type of the query that is being asked.

keyword search: just type a few keywords

structured search: a query in a language like SQL or SPARQL

natural language: a complete question, as humans typically pose it

Before we describe each of these paradigms, let us comment on some of the finer points of this categorization:

Kind of result: We also considered a (further) categorization by the kind of result but this turned out to be impractical. The kind of result usually follows from the type of query and the type of data that is being searched. Sometimes there are variations, but that usually does not affect the fundamental approach. This is briefly explained in each of the following subsections, and in more detail in the various subsections of Section 4 on *Approaches and Systems for Semantic Search*.

Search on combined data: When dealing with combined data (in the sense of Section 2.1.3), there are two prevailing search paradigms. One is basic keyword search, optionally extended with a specification of the desired type of result entities (for example, *astronauts who walked on the moon* with a restriction to entities of type *person*). This kind of search is discussed in Section 4.5. The other is structured search that is extended with a keyword search component (there are many variants for the semantics of such an extension). This kind of search is discussed in Section 4.6.

Keyword vs. natural language: Keyword search and natural language search are less clearly delineated than it may seem. For example, consider the simple query *birth date neil armstrong*. A state-of-the-art system for keyword search on text will return (on a typical corpus, say the Web) a prominent document that contains the query words in prominent position (say, the Wikipedia article of Neil Armstrong). This document will probably also contain the desired piece of information (his birth date). We classify such a system under keyword search. A state-of-the-art question answering system might understand the very same query as an abbreviated natural language query (*what is the birth date of neil armstrong*), much like a human would do when seeing the four keywords above. It would then return the corresponding answer in a human-friendly form. We classify such a system under natural language search.

An example for the opposite case would be *how to tell gzip to leave the original file*. This is clearly a natural language query. But, at the time of this writing, there happens to be a web page with exactly that title and the complete answer as content. Any state-of-the-art system for keyword search will easily find this web page, without any semantic analysis of the query whatsoever.

The bottom line is that the distinction between keyword search and natural language search is best made not by the apparent form of the query, but by the basic technique used to process the query. From this point of view, it is then usually clear to which of the two categories a given system belongs.

2.2.1 Keyword Search

Query <i>Example 1</i> <i>Example 2</i>	Keywords (typically few) <i>space flight</i> <i>apollo astronauts</i>
Result <i>Example 1</i> <i>Example 2</i>	Documents or entities (or both) that are “relevant” to the information need <i>Documents on the topic of space flight</i> <i>Astronauts from the Apollo missions</i>
Strength	Easy and quick for the user to query
Weakness	Often hard to guess the precise information need, that is, what it means to be “relevant”

This is still the most ubiquitous search paradigm, and the one we are most familiar with. All of the major web search engines use it. The user types a few keywords and gets a list of matching items in return. When the data is text, matching items are (snippets of) documents matching the keywords. When the data is a knowledge base, matching items are entities from the knowledge base. With combined data, the result is a combination of these, usually grouped by entity.

The query processing is based on matching the components of the query to parts of the data. In the simplest case, the keywords are matched literally. In more sophisticated approaches, also variants of the keywords are considered as well as variants or expansions of the whole query. This is explained in more detail in Section 4.1 on *Keyword Search on Text*. With respect to our classification, all such techniques still count as keyword search.

In contrast, natural language search tries to “understand” the query. This often means that the query is first translated to a logical form. Then that logical form is matched to the data being searched.

Recall from the discussion at the beginning of this section that even the most basic form of keyword search (which looks for literal matches

of the query words) can answer a complex natural language query, when there is a document with exactly or almost that question in, say, the title. We still consider that keyword search in this survey.

2.2.2 Structured Search

Query <i>Example</i>	Structured query languages like SPARQL <i>SELECT ?p WHERE { ?p has-profession Scientist . ?p birth-date 1970 }</i>
Result <i>Example</i>	Items from the knowledge base matching the query; the order is arbitrary or explicitly specified (using an ORDER BY clause in SPARQL) <i>Scientists born in 1970</i>
Strength	Maximally precise “semantics” = it is well-defined, which items are relevant to the query
Weakness	Cumbersome to construct queries; hard to impos- sible to guess the correct entity and relation names for large knowledge bases

Structured query languages are the method of choice when the data is inherently structured, as described in Section 2.1.2 on *Knowledge Bases*. Then even complex information needs can be formulated without ambiguity. The price is a complex query language that is not suited for ordinary users. For large knowledge bases, finding the right entity and relation names becomes extremely hard, even for expert users. Interactive query suggestions (see Section 2.3) can alleviate the problem, but not take it away.

The example query in the box above is formulated in SPARQL [2008], the standard query language for knowledge bases represented via triples. SPARQL is a recursive acronym for: SPARQL Protocol and RDF Query Language. It is very much an adaption of SQL [1986], the standard query language for databases. SQL is an acronym for:

Structured Query Language. The translation from SPARQL to SQL is discussed in Section 4.2 on *Structured Search in Knowledge Bases*.

On combined data (as discussed in Section 2.1.3), structured search requires an extension of SPARQL by a text search component. A simple realization of this is discussed in Section 4.2.1, a semantically deeper realization is discussed in Section 4.6.1.

2.2.3 Natural Language Search

Query <i>Example</i>	Natural language queries, often starting with one of the 5W1H: Who, What, Where, When, Why, How <i>Who won the oscar for best actor in 2015 ?</i>
Result <i>Example</i>	The correct answer, in human-friendly form <i>Eddie Redmayne ... and maybe some disambiguating information, like a picture, his nationality, and his birth date</i>
Strength	Most natural for humans, suitable for speech input
Weakness	Ambiguity in natural language; queries can be very complex; queries can require complex reasoning

This is the most natural form of communication for humans. In the simplest case, queries ask for an entity from a single relationship, like in the example above. More complex queries may ask for the combination of several relationships, for example:

what is the gdp of countries with a literacy rate of under 50%

A query may also ask several questions at once, for example:

what is the population and area of germany

Natural language search may also accept and correctly process keyword queries, for example: *oscar best actor 2015*. As explained at the beginning of Section 2.2, the yardstick is not the apparent form of the query but the technique used to process it.

Questions that require the synthesis of new information or complex reasoning are out of scope for this survey. For example:⁵

HAL, despite your enormous intellect, are you ever frustrated by your dependence on people to carry out your actions?

2.3 Other Aspects

Our basic classification focuses on the main aspects of a search paradigm: the kind of queries that are supported and the basic technique used to process these queries. In actual systems, a variety of other aspects can play an important role, too. We here name three particularly important and widely used aspects:

Interactivity: The search engine may provide autocompletion, query suggestions, and other means to aid the query construction. This is particularly important for semantic search. The query language may be more complex, and thus unaided query construction may be hard, even for an expert. Precise formulation of the names of entities or relations can be key to get meaningful results.

Interactivity can also help the system to get more information from the user (on the query intent), which can help result quality. For example, the query suggestions of Google steer users towards queries that the search engine can answer well.

Faceted Search: Along with the results, the search engine may provide various kinds of categories to narrow, broaden, or otherwise modify the search. Again, this is particularly important for semantic search. For complex queries and when the result is not simply a single entity or web page, one iteration may simply not be enough to get to the desired results.

Result presentation: When searching for entities, it is often useful to group the results by entity and accompany them with additional information, like an appropriate picture. When the entity was extracted from text, it can be useful to show a containing snippet of appropriate

⁵Question to the HAL 9000 computer in the movie “2001: A Space Odyssey”.

size. When the entity comes from a knowledge base, it can be useful to show some related information. When searching for entities that have a geographical location (like cities or events), it is often useful to depict those locations on a map. The map may be interactive in a number of ways. For example, hits can be represented by markers and clicking on a marker provides more detailed information or zooms in. When searching for events, they could be shown on a timeline.

Note that these extensions make sense for any data type and any search paradigm discussed in the previous sections. For example, if the results are cities, it makes sense to show them on a map, no matter how the query was formulated, and no matter whether the information came from a knowledge base or from the Semantic Web.

Several of the systems described in our main Section 4 on *Approaches and Systems for Semantic Search* implement one or several of these extensions. We provide details when we describe the respective systems. Apart from that, we do not delve deeper into these aspects in this survey. We do acknowledge though that proper user interfaces and result presentation are essential for the success of semantic search. In fact, at the time of this writing, all the major search engines already have basic features for each of the aspects discussed above.

3

Basic NLP Tasks in Semantic Search

Semantic search is about search with meaning. In text, this meaning is expressed in natural language. Even a knowledge base, where much of the meaning is implicit in the structure, has elements of natural language, for example, in the (sometimes rather long) relation names or in the object literals (which, in principle, can contain arbitrary text).

The following subsections discuss four basic techniques to capture aspects of the meaning of natural language text: POS tagging and chunking, entity recognition and disambiguation, sentence parsing, and word vectors. These four techniques should be in the toolbox of every researcher in the semantic search field.

Many of the systems described in our main Section 4 use one or more of these techniques. However, even NLP-agnostic approaches can achieve remarkable results for certain query classes. This is particularly true for keyword search on text, as discussed in Section 4.1. Still, there is no doubt that for queries above a certain complexity, natural language understanding is essential. This is discussed further in Section 6 on *The Future of Semantic Search*.

3.1 Part-of-Speech Tagging and Chunking

In *Part-of-Speech (POS) tagging*, the task is to assign to each word from a sentence a tag from a pre-defined set that describes the word's grammatical role in the sentence.

Definition 3.1. Given a sentence $s = (w_1, \dots, w_n)$ and a set of available POS tags T , POS tagging outputs a sequence t_1, t_2, \dots, t_n that assigns each word w_i a corresponding tag $t_i \in T$.

Some typical POS tags are: NN (noun), VB (verb), adjective (JJ), RB (adverb). Here is a POS-tagged example sentence using all of these:

Semantic/JJ search/NN is/VB just/RB great/JJ.

Depending on the application, POS tags of different granularity can be considered. For example, the tags may distinguish between singular (NN) and plural (NNS) nouns. Or between a regular adverb (RB), a comparative (RBR), and a superlative (RBS).

A closely related problem is that of *chunking*, sometimes also referred to as *shallow parsing*. The task of chunking is to identify and tag the basic constituents of a sentence, based on the POS-tagged words.

Definition 3.2. Given a sentence $s = (w_1, \dots, w_n)$ and a set of available chunk tags C , chunking outputs word sequences identified by triples (s_i, e_i, c_i) where s_i is the start index, e_i the end index and $c_i \in C$ the chunk type of chunk i . The chunks don't overlap and don't have to cover all of the words.

Some typical chunking tags are: NP (noun phrase), VB (verb phrase), ADJP (adjective phrase). A possible chunking of the example sentence above, using all these tags, is:

NP(Semantic/JJ search/NN) VB(is/VB) ADJP(just/RB great/JJ).

Chunking is a natural first step for both entity recognition and sentence parsing, which are discussed in the two subsections following this one.

3.1.1 Benchmarks and State of the Art

A classical benchmark is provided as part of the Penn Treebank [Marcus, Santorini, and Marcinkiewicz, 1993]. We describe it in more detail in Section 3.3 on *Sentence Parsing* below.

Both POS tagging and chunking can be solved fast and with high accuracy. For example, the well-known and widely-used Stanford POS tagger [Toutanova et al., 2003] achieves an accuracy of 97% on the Penn Treebank-3 [1999] dataset (2,499 stories from the Wall Street Journal). This is close to the accuracy achieved by human experts (which is also not perfect). Tagging speed was reported to be around 15,000 words per second, on a typical server in 2008.¹ In an experiment on a current server with Intel Xeon E5-1650 (3.50GHz) CPUs, the Stanford POS tagger was able to tag around 55,000 words per second.

An accuracy of 97% sounds impressive, but when looking at whole sentences this means that only 56% of the sentences are tagged without any error. But a fully correct tagging is important for sentence parsing; see the subsection below. Manning [2011] explores options to achieve a per-word accuracy of close to 100%.

3.1.2 Application to Short Text and Queries

For very short text, e.g., queries, the methods described here are less successful. For example, in the query *pink songs*, the word “pink” certainly refers to the pop artist and not the color. However, a typical POS tagger is not used to the telegraphic form of queries and would thus incorrectly tag “pink” as an adjective.

Hua et al. [2015] present an approach to solve variants of POS tagging and chunking for short texts. Unlike for regular text, the chunking is done before tagging. Throughout the whole process, decisions are made based on semantics, in particular, the coherence between candidate chunks and tags. This distinguishes the approach from those for larger texts where the structure of grammatically well-formed sentences plays a central role and probabilities of chains of tags determine the outcome. In a subsequent step, the approach also solves a variant

¹Reported on <http://nlp.stanford.edu/software/pos-tagger-faq.shtml>.

of the named-entity recognition and disambiguation problem that is described in the next subsection.

Short text understanding is very useful for question answering since it provides a semantic interpretation of the query. Thus, systems from Section 4.8 and Section 4.9 often include components that address this problem or variants tailored towards their particular use case.

3.2 Named-Entity Recognition and Disambiguation

Assume we are given a collection of text documents and a knowledge base. In the simplest case, the knowledge base is just a list of entities with their common name or names. Additional knowledge on these entities may be helpful for the task defined next.

The task of *Named-Entity Recognition (NER)* is to recognize which word sequences from the text documents might refer to an entity from the knowledge base. For example, in the following sentence, all word sequences referring to an entity which has its own page in the English Wikipedia (as of the time of this writing), are underlined:

*Buzz Aldrin joined Armstrong and became the second human to set foot on the Moon.*²

Definition 3.3. Given some text and a set of entity types T , NER outputs word sequences which mention a named entity. The mentions are identified by triples (s_i, e_i, t_i) where s_i is the start index, e_i the end index and $t_i \in T$ the entity's type. The entity mentions don't overlap and aren't nested.

Note that usually no knowledge base is required for NER. The task is only to identify possible entity mentions (typically proper nouns) which refer to an entity from a few classes. Most typically, these are: *person*, *location*, and *organization*. For example, the Stanford NER tagger [Finkel, Grenager, and Manning, 2005] is of this kind. If entities are linked to a knowledge base in a subsequent step, the knowledge base can be a valuable resource for NER already, e.g., in the form of a gazetteer.

²It is often arguable what exactly constitutes a *named entity*. For example, in some cases it may be desirable to tag *human* and *foot* as well.

The task of *Named-Entity Disambiguation (NED)* follows up on NER. The task of NER is just to identify the word sequences, that is, in the example above, underline them (and assign them a course type). The task of NED is to decide for each identified sequence to exactly which entity from the knowledge base it refers.

Definition 3.4. Given a knowledge base with entities E , some text and a set of possible entity mentions (s_i, e_i, t_i) from NER, the task of NED is to assign for each entity mention an entity from $E \cup \emptyset$. If no entity from E is mentioned, \emptyset should be assigned.

For example, in the sentence above, the word *Moon* could refer to any of the many moons in our solar system, or to the generic *Moon*, or to one of the many people named *Moon*. However, it is clear from the context of the sentence that the one moon from planet Earth is meant. Likewise, the word *Armstrong* could refer to many different people: Lance Armstrong, Louis Armstrong, Neil Armstrong etc. Again, the context makes it clear that Neil Armstrong is meant. The task of NED is to establish these “links” (indeed, NED is sometimes also referred to as *Named-Entity Linking*).

Also note that, in the example above, the word *Buzz* on its own could refer to a number of different entities: there are many people with that name, there is a film with that name, there is a series of video games with that name. It is part of the NER problem to find out that the entity reference in this sentence consists of the word sequence *Buzz Aldrin* and not just of the word *Buzz*.

For semantic search systems, we usually require NER and NED together. In the overview below, we therefore only consider the state of the art of research that considers both problems together.

3.2.1 Co-reference and Anaphora Resolution

It is a frequent phenomenon in natural language texts that an entity is referred to not by its name but by a placeholder word. For example, consider the following sentences:

The stalks of rhubarb are edible. Its roots are medicinally used. The leaves of the plant are toxic.

Co-reference and anaphora resolution means to identify all mentions that refer to the same entity. The first underlined word (*rhubarb*) is an entity reference in the sense above. The second underlined word (*its*) is a pronoun, which in this case refers to the rhubarb from the sentence before. The third underlined word (*the plant*) is a noun phrase, which in this context does not refer to a plant in general, but again to the rhubarb from the sentence before.

The three references together are called *co-references* because they refer to the same entity. The last two references are called *anaphora*, because they refer to an entity mentioned earlier in the text. Note that anaphora are not needed for expressiveness, but for the sake of brevity (pronouns are short) or variety (to avoid repeating the same name again and again).

There are many variations of this task depending on the context. Co-references can be within or across documents. Only references between noun-phrases can be considered or also between events described by whole sentences.

For some of the semantic search systems described in Section 4 on *Approaches and Systems for Semantic Search*, it is important that as many entity references are recognized and disambiguated as possible (high recall). In that case, anaphora resolution is just as important as NER and NED. However, in papers or benchmarks solely about NER or NED, anaphora resolution is usually not included as part of the problem statement.

A recent survey on anaphora resolution is provided in the book by Mitkov [2014]. Supervised approaches for co-reference resolution of noun-phrases are surveyed by Ng [2010].

3.2.2 Benchmarks and State of the Art

At the time of this writing, there were two active benchmarks: TAC and ERD. We briefly describe the setting and best results for each.

TAC: The *Text Analysis Conference (TAC)* is co-organized by the National Institute of Standards (NIST) and the Linguistic Data Consortium (LDC). The first TAC was held in 2008, and has since replaced the *Automatic Content Evaluation (ACE)* series, which had similar

goals and was last held in 2008. From 2009 until the time of this writing, TAC contained a track called *Entity Linking* [Ji, Nothman, and Hachey, 2014], with a new benchmark every year.³ In the benchmarks before 2014, the offsets of the word sequences that have to be disambiguated are given as part of the problem. That is, the NER part, according to our definition above, is already solved. For the NED part, an additional challenge is added in that some of the word sequences do not refer to any entity from the given knowledge base. It is part of the problem, to identify these as new entities and group them accordingly if there are several references to the same new entity. In 2014, a new end-to-end English entity discovery and linking task was introduced. This task requires to automatically extract entity mentions, link them to a knowledge base, and cluster mentions of entities not in the knowledge base.

The knowledge base used in all of the tasks was a collection of entities (800K of them in 2013) of type person, organization, or location from a dump of the English Wikipedia from October 2008. The corresponding entries from the Wikipedia infoboxes were also provided. The systems were evaluated on a mix of documents from news and posts to blogs, newsgroups, and discussion fora.

The best system for the 2012 and 2013 benchmarks (and the 2014 variant that provided the perfect mention as an input) is the MS_MLI system by Cucerzan [2012]. It achieved a variant of the F-measure of 70%.⁴ These systems are adoptions of the approach described in [Cucerzan, 2007]. The best system for the 2014 benchmark that also performs entity discovery is the system by Monahan et al., 2014.

ERD: The *Entity Recognition and Disambiguation Challenge (ERD)* was co-located with SIGIR 2014. An overview is given in [Carmel et al., 2014]. As the name of the challenge says, the benchmark comprises both

³Other tracks of TAC are discussed in subsection 4.3.5 of the Section on *Structured Data Extraction from Text*.

⁴The variant is referred to as b-cubed+ in the TAC benchmarks. It groups together all co-references in the same document into one cluster, and applies the F-measure to these clusters, not to the individual references. This avoids giving undue weight to frequent references in a document (which, by the way typical algorithms work, will either all be correct or all be wrong).

NER and NED, whereas TAC, before 2014, just asked for NED. Also different from TAC, it was not required to recognize and disambiguate entities that were not in the knowledge base.

The knowledge base for ERD was a dump of Freebase (see Table 2.2) from September 2009, restricted to entities with a corresponding page in the English Wikipedia at that same time. There were two tracks, with a different text collection each. For the long track, parts of the ClueWeb09 and ClueWeb12 collections (see Table 2.1) were used. For the short track, web search queries from various past TREC tracks were used. For both tracks, small test sets were provided for learning.

The best system in the long track was again the MS_MLI NEMO system [Cucerzan, 2014], with an F-measure of 76%. That paper also discusses (in its Section 1) the difference to the TAC benchmarks. The best system in the short track was SMAPH by Cornolti et al. [2014], with an F-measure of 69%.

3.2.3 Scale

The first large-scale NER+NED was performed by the SemTag project from Dill et al. [2003]. They recognized and disambiguated 434 million entity occurrences in 264 million web documents. Precision was estimated (from a sample) to be 82%. A relatively small knowledge base (TAP) was used, which explains the small number of recognized occurrences per document and the relatively high precision.

The largest-scale NER+NED at the time of this writing was performed by Google Research [Orr et al., 2013]. They recognized and disambiguated 11 billion entities on 800 million documents from the ClueWeb09 and ClueWeb12 collections (see Table 2.1). The knowledge base used was Freebase, and the NER+NED was reportedly “optimized for precision over recall”. Precision and recall were estimated (from a sample) with 80-85% and 70-85%, respectively.

Note that the corpus from Orr et al. [2013] is only about 3 times larger than the one used in the 10 year older study of Dill et al. [2003]. However, the number of entity occurrences is about 25 times larger. Also note that the web crawl of the Common Crawl project from August 2014 contained around 2.8 billion web pages (200 TB), which is only

about 3 times larger than both of the ClueWeb datasets together. In 2015, Google revealed that it knows over 30 thousand trillion different URLs on the Web. However, only a fraction of these point to textual content that is actually useful for search. Also, many URLs point to similar content. As of 2015, the number of distinct web pages indexed by Google is estimated to be around 50 billion.⁵

The bottom line is that web-scale NER+NED with a large general-purpose knowledge base is feasible with good (but still far from perfect) precision and recall.

3.3 Sentence Parsing

The goal of sentence parsing is to identify the grammatical structure of a sentence. There are two kinds of representations that are widely used: the *constituent parse* and the *dependency parse*. Both parses can be viewed as a tree. In a constituent parse, the sentence is first split, e.g., into a *subject noun phrase* (NP) and a *predicate verb phrase* (VP), which are then recursively split into smaller components until the level of words or chunks is reached.

Definition 3.5. A constituent parse of a sentence consists of a tree with the individual words as its leaves. Labels of internal nodes represent grammatical categories of the word sequences corresponding to their subtree, e.g., *noun phrase* (NP), *verb phrase* (VP), *subordinate clause* (SBAR), or *independent clause* (S). These can be nested recursively. The root of the tree is usually labeled *S*.

In a dependency parse, each word in the sentence depends on exactly one other word in the sentence, its head; the root of the tree points to the main verb of the sentence.

Definition 3.6. A dependency parse of a sentence is a tree⁶ with individual words as nodes. Nodes are connected via directed edges from the

⁵Source: <http://www.worldwidewebsize.com>, who extrapolate from the number of hits for a selection of queries.

⁶This is often also referred to as a dependency *graph*, but for all practical purposes it can be considered a tree.

governor (head) to the *dependent*. Each node has exactly one governor. Edges can be labeled with the grammatical relationship between the words. The main verb of the sentence has an artificial *ROOT* node as its head.

For example, the sentence from above has the following constituent and dependency parse, respectively. For the constituent parse, the tree structure is shown via nested parentheses labeled with constituent type. The dependency parse only shows unlabeled arcs and no *ROOT* node.

$S(NP((Semantic) (search)) VP(VB(is) ADJP((just) (great))))$.

$Semantic \leftarrow search \leftarrow is \rightarrow great \rightarrow just$.

From a linguistic perspective, the two types of grammars are almost equivalent [Gaifman, 1965]. Indeed, many widely used parsers (including the Stanford parser referred to below) produce one type of parse from which they can easily derive the other type of parse.

3.3.1 Semantic Role Labeling

Although this section is about sentence parsing, let us also briefly discuss *Semantic Role Labeling (SRL)* at this point. SRL goes beyond sentence parsing in that it also assigns “roles” to the semantic arguments of a predicate or verb of a sentence. For example, consider the following two sentences:

John gives the book to Mary.

John is given the book by Mary.

Both have the same (structure of the) constituent parse tree. But the role of John with respect to the verb *give* is different: in the first sentence, he is the one who gives (his role is *giver*), in the second sentence, he is the one who is given (his role is *recipient*).

Semantic Role Labeling looks very relevant for semantic search. For example, for the query *who was given the book*, the correct answer is different for each of the two sentences above. However, most semantic search systems nowadays work with (surprisingly) shallow linguistic technology. Many do not even use sentence parsing, and none of the systems described in this survey uses SRL.

There are two apparent reasons. One reason is that semantic search is still in its infancy, with major challenges still to overcome even for relatively basic search scenarios that do not involve any natural language processing or only a relatively shallow one. The other reason is that natural language processing has not yet reached the necessary level of sophistication in order to be unreservedly useful for improving search results. For a look further ahead see Section 6 on *The Future of Semantic Search*.

3.3.2 Benchmarks and State of the Art

Datasets: A benchmark for sentence parsing (that is, a text annotated by the correct parse for each sentence) is referred to as a *Treebank*. The most widely used Treebank is the Penn Treebank [Marcus, Santorini, and Marcinkiewicz, 1993]. It comprises 2,499 articles from the Wall Street Journal from 1989 (about 44K sentences with about 1M words), annotated for syntactic structure by human experts. The articles are split into 25 sections with the convention to use sections 2-21 as a training set and section 23 as test set. Remaining sections can be used as development set. The Penn Treebank also contains the POS-tagged Brown corpus (carefully selected English text from 15 genres, about 1M words, from 1961). There are also two follow-up benchmarks, called Penn Treebank-2 [1995] and Penn Treebank-3 [1999].

More recent English Treebanks are *Ontonotes 5.0* by Hovy et al. [2006], which also contains articles from the Wall Street Journal, and the *Google Web Treebank* by Petrov and McDonald [2012], which consists of annotated sentences from the Web. However, most English parsers are still trained and evaluated on the Penn Treebank.

CoNLL 2007 Shared Task: This task [Nivre et al., 2007] featured two tracks on dependency parsing: one *multilingual* (with an English sub-track), and one called *domain adaptation* (learn from one domain, test on another). Two standard metrics emerged from that task: the *Labeled Attachment Score (LAS)* and the *Unlabeled Attachment Score (UAS)*. Both are percentages, with 100% being a perfect result. UAS measures the degree to which the structure is correct (head and arcs). LAS also takes into account the correctness of the dependency labels.

The best system in the English part of the multilingual track achieved 89.6% LAS and 90.6% UAS. The best system in the domain adaptation track achieved 81% LAS and 83.4% UAS.

SANCL 2012 Shared Task: SANCL [Petrov and McDonald, 2012] is the name of the Workshop on Syntactic Analysis of non-canonical Language. Parsers were trained on the Penn Treebank (sections 2-21, about 30K sentences), but evaluated on the Google Web Treebank. Both dependency and constituent parsers were evaluated. The Stanford parser achieved 80.7% precision and 79.6% recall in constituent mode, and 83.1% LAS and 87.2% UAS in dependency mode. The best constituent parser achieved 84.3% precision and 82.8% recall.

Socher et al. [2013] describe recent improvements to the Stanford parser, and compare it to other state-of-the-art parsers. F-measures between 85% and 92% are achieved.

It is noteworthy that good parsers achieve about equal figures for precision and recall, which is why often only the F-measure is reported. Already the early Charniak parser [Charniak, 2000] achieved both precision and recall of about 90% (on a relatively simple benchmark).

Note that with close to perfect chunking alone (which is a relatively simple task) one already gets around 50% recall (all the minimal constituents are correct) and close to perfect precision, that is, around 75% F-measure. But such a “parsing” would be quite useless for semantic search or information extraction, where it is important to detect which items of a sentence “belong together” semantically. Bastings and Sima'an [2014] therefore introduce an alternative measure, called FREVAL. This measure weighs each component by its height in the parse tree (a leaf has height 1, the root has the largest height). Mistakes in the larger components then incur a higher penalty in the score. Using this measure, they report only 35% to 55% F-measure for current state-of-the-art parsers. Indeed, these figures better reflect our own experience of the limited use of sentence parsing for semantic search, than the close to 90% achieved in the standard measures.

A further problem is that parsers perform worse on long sentences than on short sentences. For example, Klein and Manning [2002] report a drop in F-measure from about 90% for sentences with 10 words to

about 75% for sentences with 40 words. Unfortunately, much of the information in a text is often contained in long (and often complex) sentences. This is exacerbated by the fact that available parser models are usually trained on newswire text but applied to web-like text, which is more colloquial and sometimes ungrammatical (see the results of SANCL 2012 shared task above).

3.3.3 Performance and Scale

In terms of F-measure, the Charniak constituent parser achieves the state-of-the-art result at 92% and claims about 1 second per sentence [McClosky, Charniak, and Johnson, 2006; Charniak, 2000]. The recursive neural network (constituent) parser from Socher et al. [2013] needs about 0.8 seconds per sentence, and achieves 90% F-measure on the Penn Treebank. Recently, parsers have improved parsing times considerably while maintaining or improving state-of-the-art quality. The greedily implemented shift-reduce based constituent parser that is part of the Stanford CoreNLP toolkit [Manning et al., 2014] achieves comparable 88.6% F-measure but is about 30 times as fast (27 ms per sentence). A recent neural network based dependency parser [Chen and Manning, 2014] can process about 650 sentences per second (1.5 ms per sentence) and produce state-of-the-art results (89.6% LAS and 91.8% UAS on the Penn Treebank with Stanford dependencies). spaCy⁷ is the currently fastest greedy shift-reduce based parser, which can process about 13K sentences per second (0.08 ms per sentence) with state-of-the-art performance (91.8% UAS on the Penn Treebank). A recent comparison of parsers is given by Choi, Tetreault, and Stent [2015].

3.4 Word Vectors

Word vectors or *word embeddings* represent each word as a real-valued vector, typically in a space of dimension much lower than the size of the vocabulary. The main goal is that semantically similar words should have similar vectors (e.g., with respect to cosine similarity). Also, word vectors often have linear properties, since they are usually obtained by

⁷<https://spacy.io/>

(implicit or explicit) matrix factorization. For example, the methods described below will produce similar vectors for *queen* and *king*, because they are both monarchs, as well as similar vectors for *queen* - *woman* + *man* and *king*. Word vectors are also popular as a robust representation of words used as input to machine learning algorithms.

Research on word vectors has a long history in natural language processing dating back to a famous statement by John Rupert Firth in 1957: *You shall know a word by the company it keeps*. Indeed, words that occur in similar contexts are likely to be similar in meaning. This implies that word vectors can be learned in an unsupervised fashion from a huge text corpus, without additional knowledge input. In the following, we discuss the main techniques, extensions to text passages, and the most popular benchmarks.

Applying word vectors has recently gained interest. In this survey many of the approaches in Section 4.8 on *Question Answering on Knowledge Bases* use it as part of the input to machine learning algorithms in order to provide a notion of (semantic) synonyms. It is also part of the future work planned for many recent systems, for example, for Joshi, Sawant, and Chakrabarti [2014] in Section 4.9 on *Question Answering on Combined Data*.

3.4.1 Main Techniques

The straightforward approach is to build a word-word co-occurrence matrix [Lund and Burgess, 1996], where each entry counts how often the two words co-occur in a pre-defined context (in the simplest case: within a certain distance of each other). A row (or column) of the matrix can then be considered as a (huge but sparse) word vector. From there, a low-dimensional dense embedding can be obtained via matrix factorization techniques. For example, using principal component analysis (PCA, typically computed from an eigenvector decomposition) or non-negative matrix factorization (NMF, typically computed with a variant of the EM algorithm) [Lee and Seung, 2000]. There are many variations of this basic approach; for example, the co-occurrence matrix can be row-normalized, column-normalized, or each entry can be replaced by its positive pointwise mutual information.

Explicit semantic analysis (ESA) [Gabrilovich and Markovitch, 2007] represents a word as a vector of weighted Wikipedia concepts. The weight of a concept for a word is the tf-idf score of the word in the concept's Wikipedia article. The resulting word vectors are often sparse, because each concept article contains only a small subset of all possible words. By construction, longer text passages can be represented by the sum of the word vectors of the contained words. The resulting vector is then supposed to be a good representation of what the text "is about". Like PCA and NMF, ESA can be combined with standard ranking techniques (like BM25) to improve retrieval quality in keyword search on text.

Word2vec [Mikolov et al., 2013a; Mikolov et al., 2013b] computes (dense) word vectors using a neural network with a single hidden layer. The basic idea is to use the neural network for the following task: given a current word w_i , predict the words w_{i+c} occurring in its context (a window around w_i , e.g., positions $-2, -1, +1, +2$). The network is trained on an arbitrary given text corpus, with the goal of maximizing the product of these probabilities. Once trained, the word vectors can be derived from the weights of the intermediate layer. Interestingly, Levy and Goldberg [2014] could show that word2vec implicitly performs a matrix factorization of the word-context matrix. The major advantage over the explicit matrix factorization techniques from above is in space consumption and training speed; see the next but one paragraph.

Glove [Pennington, Socher, and Manning, 2014] is a log-bilinear regression model that, intuitively, is trained to predict word co-occurrence counts. The model effectively performs a factorization of the log co-occurrence count matrix [Levy, Goldberg, and Dagan, 2015]. Experiments show that it performs similarly to word2vec; see the next paragraph. It is also fast to train, but requires a co-occurrence count matrix as input.

Levy, Goldberg, and Dagan [2015] perform an extensive comparison of the many approaches for word vectors, including word2vec, Glove and co-occurrence based methods. They also give valuable advice on how to choose hyperparameters for each model. In their experiments, none of the techniques consistently outperforms others. Experiments

also show that hyperparameters have a huge impact on the performance of each model, which makes a direct comparison difficult. However, they report significant performance differences on a corpus with 1.5 billion tokens, in particular: half a day versus many days for the training phases of word2vec vs. Glove, and an unfeasibly large memory consumption for the explicit factorization methods.

3.4.2 Extensions to Text Passages

The straightforward approach to obtain a low-dimensional vector of the kind above for an arbitrary text passage is to sum up or average over the vectors of the contained words. This works well in some applications, but can only be an approximation because it completely ignores word order.

Le and Mikolov [2014] have extended word2vec to compute vectors for paragraphs and documents. The vector is learned for the given passage as a whole, and not just statically composed from individual word vectors. On a sentiment analysis task, the approach beats simple composition methods (as described in the previous paragraph) as well as classical supervised methods (which do not leverage external text).

Two related problems are *paraphrasing* and *textual entailment*, where the task is to determine for two given pieces of text, whether they mean the same thing or whether the first entails the second, respectively. For example, does *John go to school every day* entail *John is a student*? Learning-based methods for paraphrasing and textual entailment are discussed in Section 8.1 of the survey by Li and Xu [2014].

3.4.3 Benchmarks

There are two popular problems that allow an intrinsic evaluation of word vectors. For each problem, several benchmark datasets are available.

Word similarity: This is a ranking task. Word pairs must be ranked by how similar the two words are. For example, the words of the pair (*error*, *mistake*) are more similar than (*adventure*, *flood*). Benchmarks contain word pairs with human-judged graded similarity scores, often

retrieved via crowdsourcing. The final quality is assessed by computing rank correlation using Spearman’s ρ . Some relevant benchmarks are:

WordSim353 [Finkelstein et al., 2002]: 353 word pairs

SimLex-999 [Hill, Reichart, and Korhonen, 2015]: 999 word pairs

MEN [Bruni, Tran, and Baroni, 2014]: 3000 word pairs

Rare words [Luong, Socher, and Manning, 2013]: 2034 word pairs

Crowdsourcing benchmark [Radinsky et al., 2011]: 287 word pairs

For a recent performance comparison we refer to the experiments done by Levy, Goldberg, and Dagan [2015]. Typical rank correlations are between .4 and .8, depending on the dataset and the model.

Word analogy: Analogy questions are of the form “*saw* is to *sees* as *returned* is to ?” and the task is to fill in the missing word (*returns*). More formally, the task is: given words a , a^* and b find the word b^* such that the statement “ a is to a^* as b is to b^* ” holds. One variant of this task addresses syntactic similarities, as in the example above. The other variant focuses on semantic similarities as in “*Paris* is to *France* as *Tokyo* is to ?” (*Japan*). To solve this task, simple vector arithmetic is used. Most prominently:

$$\arg \max_{b^* \in V \setminus \{a, a^*, b\}} \cos(b^*, a^* - a + b)$$

where V is the vocabulary. Levy, Goldberg, and Dagan [2015] improved on that function, essentially by taking the logarithm. The resulting function is called *3CosMul*:

$$\arg \max_{b^* \in V \setminus \{a, a^*, b\}} \frac{\cos(b^*, a^*) \cdot \cos(b^*, b)}{\cos(b^*, a) + \epsilon}$$

Notable benchmarks are from Microsoft Research [Mikolov, Yih, and Zweig, 2013], which consists of 8,000 questions focusing on syntactic similarities, and Google [Mikolov et al., 2013b], which consists of 19,544 questions for syntactic as well as semantic similarities. The evaluation measure is the percentage of words b^* that were correctly predicted. Again, we refer to the experiments from Levy, Goldberg, and Dagan [2015] for a recent comparison. Current models answer about 55% to 69% of these questions correctly.

4

Approaches and Systems for Semantic Search

This is the core section of this survey. Here we describe the multitude of approaches to and systems for semantic search on text and knowledge bases. We follow the classification by data type and search paradigm from Section 2, depicted in Figure 1.1. For each of the nine resulting subgroups, there is a subsection in the following. Each of these subsections has the same structure:

Profile ... a short characterization of this line of research

Techniques ... what are the basic techniques used

Systems ... a concise description of milestone systems or software

Benchmarks ... existing benchmarks and the best results on them

We roughly ordered the sections historically, that is, those scenarios come first, which have been historically researched first (and the most). Later sections correspond to more and more complex scenarios, with the last one (Section 4.9 on *Question Answering on Combined Data*) being the hardest, with still relatively little research to date. Also, approaches and systems of the later section often build on research from the more fundamental scenarios from the earlier sections. For example, almost any approach that deals with textual data uses standard data structures and techniques from classical keyword search on text.

4.1 Keyword Search in Text

Data	Text documents, as described in Section 2.1.1
Search	<p>Keyword search, as described in Section 2.2.1</p> <p>This is classical full-text search: the query is a sequence of (typically few) keywords, and the result is a list of (excerpts from) documents relevant to the query</p> <p><i>Methods aimed at a particular entity or list of entities are addressed in Section 4.5</i></p>
Approach	Find all documents that match the words from the query or variants/expansions of the query; rank the results by a combination of relevance signals (like prominent occurrences of the query words in the document or occurrences in proximity); learn the optimal combination of these relevance signals from past relevance data
Strength	Easy to use; works well when document relevance correlates well with basic relevance signals
Limitation	Is bound to fail for queries which require a match based on a deeper understanding (of the query or the matching document or both), or which requires the combination of information from different sources

This is the kind of search we are all most familiar with from the large web search engines: you type a few keywords and you get a list of documents that match the keywords from your query, or variations of them.

A comprehensive treatment of this line of research would be a survey on its own. We instead provide a very brief overview of the most important aspects (Section 4.1.1), widely used software which implements the state of the art (Section 4.1.2), and an overview over the

most important benchmarks in the field and a critical discussion of the (lack of major) quality improvements over the last two decades (Section 4.1.3).

4.1.1 Basic Techniques

With respect to search quality, there are two main aspects: the *matching* between a keyword query and a document, and the *ranking* of the (typically very many) matching documents. We do not cover performance issues for keyword search on text in this survey. However, Section 5.2 discusses various extensions of the inverted index (the standard indexing data structure for keyword search on text) to more semantic approaches.

Basic techniques in matching are: lemmatization or stemming (*houses* \rightarrow *house* or *hous*), synonyms (*search* \leftrightarrow *retrieval*), error correction (*algorithm* \leftrightarrow *algorithm*), relevance feedback (given some relevant documents, enhance the query to find more relevant documents), proximity (of some or all of the query words) and concept models (matching the *topic* of a document, instead of or in addition to its words). A recent survey on these techniques, cast into a common framework called *learning to match*, is provided by Li and Xu [2014].

Basic techniques in ranking are either query-dependent ranking functions, like BM25 (yielding a score for each occurrence of a word in a document) and language models (a word distribution per document), or query-independent popularity scores, like PageRank (yielding a single score per document). Hundreds of refinements and signals have been explored, with limited success; see the critical discussion in the benchmark section below. The most significant advancement of the last decade was the advent of *learning to rank* (*LTR*): this enables leverage of a large number of potentially useful signals by learning the weights of an optimal combination from past relevance data. See [Liu, 2009] for a survey with a focus on applicable machine learning techniques. We discuss applications of LTR to other forms of semantic search in Section 5.1 on *Ranking*.

4.1.2 Popular State-Of-The-Art Software

Researchers have developed countless systems for keyword search on text. A list is beyond the scope of this article, and bound to be very incomplete anyway. Instead, we focus on open-source software and prototypes that are widely used by the research community. Each of the systems below provides basic functionality like: incremental index updates (adding new documents without having to rebuild the whole index), fielded indices (to store arbitrary additional information along with each document), distributed processing (split large text collections into multiple parts, which are then indexed and queried in parallel), standard query operators (like: conjunction, disjunction, proximity), and multi-threading (processing several queries concurrently). Also, each of the systems below is used as basis for at least one system for more complex semantic search, which are described in one of the following sections.

There are several studies comparing these systems. For a quality comparison of some of these, see [Armstrong et al., 2009a]. For a performance comparison, see [Trotman et al., 2012].

Apache's Lucene¹ is the most widely used open-source software for basic keyword search. It is written in Java and designed to be highly scalable and highly extensible. It is the most used software in commercial applications. Lucene provides built-in support for some of the basic matching and ranking techniques described in Section 4.1.1 above: stemming, synonyms, error correction, proximity, BM25, language models.

Indri² is written in C++. It is a general-purpose search engine, but particularly used for language-model retrieval. Terrier³ is written in Java, and provides similar functionality as Indri.

MG4J [Boldi and Vigna, 2005] is written in Java. It makes use of *quasi-succinct* indexes, which are particularly space-efficient and enable particularly fast query processing also for complex query operators. MG4J supports fielded BM25, which is used by various of the

¹<http://lucene.apache.org>

²<http://www.lemurproject.org/indri>

³<http://terrier.org>

approaches described in Section 4.5 on *Keyword Search on Combined Data*.

4.1.3 Benchmarks

The classical source for benchmarks for keyword search on unstructured text is the annual Text Retrieval Conference (TREC) series [Voorhees and Harman, 2005], which began in 1992.⁴ TREC is divided into various so-called *tracks*, where each track is about a particular kind of retrieval task. Each track usually runs over a period of several years, with a different benchmark each year. Each benchmark consists of a document collection, a set of queries, and relevance judgments for each query.⁵

Keyword search on text documents was considered in the following tracks: *Ad-hoc* (1992 - 1999, keyword search on the TIPSTER⁶ collection), *Robust* (2003 - 2005, hard queries from the ad-hoc track), *Terabyte* (2004 - 2006, much larger document collection than in previous tracks), and *Web* (1999 - 2004 and 2009 - 2014, web documents).

Armstrong et al. [2009a] and Armstrong et al. [2009b] conducted an extensive comparative study of the progress of ad-hoc search over the years. Systems were compared in two ways: (1) by direct comparison of different results from different papers on the same (TREC ad-hoc) benchmarks, and (2) by a comparison across benchmarks using a technique called *score standardization*. Their surprising conclusion from both studies is that results for ad-hoc search have not improved significantly since 1998 or even earlier. New techniques were indeed introduced, but the evaluations were almost always against weak baselines, instead of against the best previous state-of-the-art system.

Viewed from a different perspective, this study merely confirms a typical experience of information retrieval researchers regarding keyword search. The shortcomings are clear, and promising new ideas

⁴When researching proceedings, it helps to know that the first 9 TREC conferences, from 1992 to 2000, are referenced by number: TREC-1, ..., TREC-9. Starting from 2001, they are referenced by year: TREC-2001, TREC 2002, ...

⁵For the later (very large) collections, only partial relevance judgments (for the top documents from each participating system) were available. This is called *pooling*.

⁶The TIPSTER collection comprises news articles, government announcements, and technical abstracts.

spring to mind relatively quickly. But a comprehensive and honest evaluation of any single idea over a large variety of queries is often sobering: the results for some queries indeed improve (usually because relevant documents are found, which were not found before), while the results for other queries deteriorate (usually because of lower precision). Often, the two opposing effects more or less balance out, and it is mostly a matter of careful parameter tuning to get a slight improvement out of this.

A real improvement was brought along by the learning to rank approach, discussed briefly in Section 4.1.1 above. With learning to rank, a large number of potentially useful signals can be combined, and the best “parameter tuning” can be learned automatically from past relevance data. Indeed, the winners of the last three TREC Web Tracks are all based on this approach.

In absolute terms, results remained relatively weak however, with typical $nDCG@20$ values of around 30%. This makes it all the more reasonable to go beyond this simple form of keyword search and aim at deeper forms of understanding, which is exactly what the approaches described in the following sections do.

4.2 Structured Search in Knowledge Bases

Data	A knowledge base, as described in Section 2.1.2
Search	Structured search, as described in Section 2.2.2 The query is from a language like SPARQL; the result is a list of matching items from the knowledge base; the order is arbitrary or explicitly specified
Approach	Store the knowledge base in a standard RDBMS and rewrite queries to SQL; or use a dedicated index data structure and query engine
Strength	Expert searches with a precise semantics; the canonical back end for any service that involves non-trivial queries to a knowledge base
Limitation	Query formulation is cumbersome, especially for complex queries; finding the right entity and relation names becomes very hard on large knowledge bases; the amount of information contained in knowledge bases is small compared to the amount of knowledge contained in text

Structured search in knowledge bases is not so much a technique for semantic search on its own, but rather a basic building block for all approaches that work with one or more knowledge bases.

4.2.1 Basic Techniques

There are two main approaches to storing a knowledge base: in a standard relational database management system (RDBMS), or in a system dedicated to storing knowledge as collections of triples and hence often called *triple store*. Both approaches are widely used. The design and implementation of a typical triple store is described in Section 4.2.2 below.

When the knowledge base is stored in an RDBMS and the query language is SPARQL, queries can be translated to equivalent SQL queries.

A complete translation scheme is described in [Elliott et al., 2009].⁷ When the data is stored in an RDBMS using a non-trivial schema (that is, not just one big table of triples), a mapping is needed to specify how to make triples out of this data. For this mapping, R2RML [2012] has emerged as a standard. Given such a mapping, generating a SQL query that can be executed as efficiently as possible becomes a non-trivial problem [Unbehauen, Stadler, and Auer, 2013].

Traditional RDBMSs store their data row oriented, that is, the items from one row are contiguous in memory. This is advantageous when retrieving complete rows via direct access (e.g., via their key). When storing a knowledge base in an RDBMS, column orientation is the layout of choice. This is because typical SPARQL queries require scans of very long runs of entries for one attribute. For example, to find all people born in a given city, we need to determine all triples with that city as their object. Also, these columns are typically highly compressible. For the example just given, there will be long runs of triples with the same city (if sorted by object). A simple run-length encoding then saves both space and query time. A recent survey on column-oriented databases aka column stores (with focus on efficiency) is provided by Abadi et al. [2013].

The list of systems and benchmarks in Sections 4.2.2 and 4.2.3 below focuses on systems that explicitly support SPARQL. There are two main aspects when comparing these systems: their performance and which features they support.

Performance

It appears that dedicated triples stores have an advantage over RDBMS-based systems. Dedicated triple stores can use index data structures that are tailored to sets of triples (in particular, exploiting the high repetitiveness and hence compressibility involved, see above). Similarly, they can use query optimizers that exploit the structure of

⁷The SPARQL features ASK, CONSTRUCT, and DESCRIBE are treated specially, since they can only be approximated in SQL. They are not essential for the expressiveness of SPARQL, however.

typical SPARQL queries.⁸ It turns out, however, that RDBMS-based approaches can still be superior, especially for complex queries, because of their more mature query-optimizer implementations. This is briefly discussed in the benchmark subsection below. Query planning and optimization are a research topic of their own, and we refer the interested reader to [Schmidt, Meier, and Lausen, 2010].

Features

All the widely used systems below support the full SPARQL standard. Research prototypes often focus on SELECT queries, details below. Other features, which some but not all of the systems provide, are:

Reasoning: support for reasoning, e.g., using OWL or RDFS; this is the topic of Section 5.4 on *Inference*.

Web API: query or modify the database via HTTP.

Exchangeable back end: plug in different back ends; in particular, allow the choice between a dedicated triple store and an RDBMS.

Full-text search: support for keyword search in objects which are string literals; here is an example using the syntax from Virtuoso’s keyword search extension (the prefix *bif* stands for built-in function and prefixes for the other relations are omitted):

```
SELECT ?p WHERE {
  ?p has-profession Astronaut .
  ?p has-description ?d .
  ?d bif:contains "walked AND moon" }
```

Note that already standard SPARQL enables regular-expression matching of entity names via the *FILTER regex(...)* operation. In principle, regular expressions can simulate keyword queries, but not very practically so. For example, a string literal matches the two keywords *w1* and *w2* if it matches one of the regular expressions *w1.*w2* or *w2.*w1*.

⁸From a more general perspective, such special-purpose databases are often called *NoSQL* (acronym for “non (relational) SQL”, sometimes also interpreted as “not only SQL”). Another example of a NoSQL database is Google’s *BigTable*, which supports database-like queries on extremely large amounts of data that may be stored distributed over thousands of machines.

Note that entity names are also string literals. This simple kind of search is hence also useful when the exact name of the entity is not known, or for long names. For example, the entity name *Barack Hussein Obama* would be found with the keyword query "barack AND obama".

4.2.2 Systems

The three most widely used systems at the time of this writing are (in chronological order of the year the system was introduced): Virtuoso⁹, Jena¹⁰, and Sesame [Broekstra, Kampman, and Harmelen, 2002].

All three provide all of the features listed above. Virtuoso is written in C, Jena and Sesame are written in Java. Virtuoso is different in that it is also a full-featured RDBMS; in particular, it can run with its own RDBMS as back end. For a performance comparison, see the benchmarks below.

Traditional database companies, like Oracle or MySQL, have also started to provide support for triple stores and SPARQL queries. However, at the time of this writing, they still lack the breadth of features of systems like Virtuoso, Jena, or Sesame.

Details of the implementation of a dedicated triple store and SPARQL engine are described in [Neumann and Weikum, 2009; Neumann and Weikum, 2010], for a system called RDF-3X. The software is open source. RDF-3X supports SELECT queries with the most important modifiers and patterns.¹¹ RDF-3X builds an index for each of the six possible permutations of a triple (SPO, SOP, OPS, OSP, POS, PSO, where S = subject, P = predicate, O = object). This enables fast retrieval of the matching subset for each part of a SPARQL query. Join orders are optimized for typical SPARQL queries, including star-shaped (all triples have the same variable as their subject) and paths (the object of one triple is the subject of the next). Query plans are ranked using standard database techniques, like estimating the cost

⁹<http://virtuoso.openlinksw.com>

¹⁰<https://jena.apache.org>

¹¹Supported patterns: OPTIONAL and FILTER. Supported modifiers: ORDER BY, DISTINCT, REDUCED, LIMIT, and OFFSET. Not supported: ASK, DESCRIBE, and CONSTRUCT queries.

via histogram counts. The authors provide a performance evaluation, where RDF-3X is faster than two column-store RDBMs (MonetDB and PostgreSQL) on a variety of datasets (including BTC'09 and UniProt from Tables 2.3 and 2.2). This is inconsistent with the results from the Berlin SPARQL benchmark, discussed below, where an RDBMs (Virtuoso) wins when the data is very large.

Bast et al. [2014a] provide a system for the incremental construction of tree-like SPARQL queries. The system provides context-sensitive suggestions for entity and relation names after each keystroke. The suggestions are ranked such that the most promising suggestions appear first; this ranking is discussed in more detail in Section 5.1.4. As of this writing, an online demo for Freebase (see Table 2.2) is available: <http://freebase-easy.cs.uni-freiburg.de>. The demo also addresses the challenge of providing unique and human-readable entity names.¹²

SIREn [Delbru, Campinas, and Tummarello, 2012] uses an inverted index (Lucene) to support star-shaped SPARQL queries (with one entity at the center), where predicate and relation names can be matched via keyword queries. We describe the index in more detail in Section 5.2.1 on *Using an Inverted Index for Knowledge Base Data*.

4.2.3 Benchmarks

The Berlin SPARQL Benchmark [Bizer and Schultz, 2009] is modeled after a real use case: a consumer looking for a product on an e-commerce website. 12 generic queries are chosen to model the SPARQL queries sent to the back end during such a session. The queries are parameterized, e.g., by the type of product that the consumer is looking for initially. The benchmark demands that the queries are asked in sequence, with multiple sequences being asked concurrently, again as in a real setting. The dataset is modeled after a real set of products (with various features and textual descriptions) and is synthetically generated, with an arbitrary, given size.

¹²The entity names from Freebase are not unique, and the identifiers are alphanumeric strings. In contrast, for example, Wikipedia has human-readable unique identifiers for each of its entities.

Bizer and Schultz [2009] compare a large variety of systems: explicit triple stores (including: Jena, Sesame, and Virtuoso with its own triple store back end), and SPARQL-to-SQL rewriters using an RDBMS (including: MySQL and Virtuoso with its own RDBMS back end). Dataset sizes used were 1M, 25M, and 100M triples. No single system came out as the clear winner. However, for the largest datasets (100M), the best RDBMS-based approach (Virtuoso) was about 10 times faster on average than the best dedicated triple store. The authors attribute this to the more mature query optimizers of established RDBMS systems. It is noted that the SPARQL-to-SQL rewriting takes up to half the time.

The DBpedia SPARQL Benchmark [Morsey et al., 2011] is a generic benchmark that aims at deriving *realistic* SPARQL queries from an arbitrary given query log for an arbitrary given knowledge base. In particular, 25 query templates are derived for the DBpedia dataset (see Table 2.2). Their evaluation confirms the performance differences from previous benchmarks, notably Bizer and Schultz [2009], except that the performance differences are even larger with realistic data and queries.

4.3 Structured Data Extraction from Text

Data	Text documents, as described in Section 2.1.1 This includes web documents with markup that helps to identify structure in the data
Search	The main purpose of the systems described in this section is to extract structured information from text; the search is then an add-on or left to systems as discussed in Section 4.2 on <i>Structured Search in Knowledge Bases</i>
Approach	Extract structured data from text; store in a knowledge base or reconcile with an existing one; an alternative for very simply structured queries is to translate them to suitable keyword queries
Strength	Make the vast amounts of structured data contained in text documents accessible for structured search
Limitation	Extraction with high precision and recall is hard; reconciling extracted information in a single knowledge base is hard; some information is hard to express in structured form

A large part of the world's information is provided in the form of natural language text, created for humans. Large amounts of this information could be naturally stored (and queried) in structured form.

4.3.1 Basic Techniques

We distinguish three kinds of approaches to access structured data contained in text documents: relationship extraction from natural language text, extraction of tables or infoboxes, and knowledge base construction. In a sense, the approaches build on each other, which is why we describe them in this order.

For each of the three approaches, we describe the state-of-the-art systems and performance in Sections 4.3.2 - 4.3.4. For a holistic overview of the whole field of information extraction from text we refer to the excellent survey from Sarawagi [2008].

Relationship Extraction from Natural Language Text

Relationship extraction aims at extracting subject-predicate-object tuples from a given collection of natural language text. Consider the following sentence from Wikipedia:

Aldrin was born January 20, 1930, in Mountainside Hospital, which straddles both Glen Ridge and Montclair

In basic relationship extraction, the searched relation is part of the input. For example, extract all triples for the *place of birth* relation from a given text. For the sentence above such a triple would be:

Buzz Aldrin place of birth Glen Ridge

The subject and object may or may not be linked (one also says: *grounded*) to a matching entity from a given knowledge base (in the example they are: we use the names from an – imaginary in this case – knowledge base, not from the original sentence). Since the relation was given, the predicate is easily grounded. Depending on the verb, there may be multiple objects (in the example, there is just one).

Banko et al. [2007] introduced *Open Information Extraction* (OIE), where the goal is to extract as many tuples as possible (for any relation) from the given text. For the example sentence above, a typical OIE system would extract:

Aldrin was born Glen Ridge

Unlike for the triple above, the subject and, especially, the predicate are not grounded, but are simply expressed using words from the sentence.

Specialized Extraction

Web documents often contain additional structure in the form of markup for some of the contents. Two notable such sub-structures

are *tables* and Wikipedia *infoboxes*. Tables are interesting because a lot of structured information contained in text is formatted as tables. Balakrishnan et al. [2015] report that they have indexed over a hundred million HTML tables that contain interesting structured data.¹³ Infoboxes are interesting because Wikipedia covers a lot of general-purpose knowledge with high quality. In Section 4.3.3 below, we discuss several systems developed for these sub-structures.

There is also vast literature on domain-specific extraction, in particular, for the life sciences. For example, extract all pairs of proteins (subject and object) that interact in a certain way (predicate) from a large collection of pertinent publications. The main challenge for such systems is domain-specific knowledge (e.g., the many variants how protein names are expressed in text), which is beyond the scope of this survey.

Knowledge Base Construction

Basic extraction processes, as described in the previous two subsections, yield a (typically very large) collection of elements of structured data, often triples. To obtain a knowledge base, as described in Section 2.1.2, two challenging steps are still missing: entity resolution and knowledge fusion, which we briefly explain here.

For *entity resolution*, sometimes also called entity de-duplication, strings referring to the same entity must be mapped to a unique identifier for that entity. For example, the extraction process might yield the two triples:

Buzz Aldrin *born in* *Glen Ridge*
Aldrin *born in* *Montclair*

Here, the two subjects are different strings but refer to the same entity. Depending on the extraction process, this might also happen for the predicates.

For *knowledge fusion*, different triples might contain conflicting or complementary information, which needs to be resolved or unified. For the two triples above, both provide correct information in a sense (the

¹³Note that HTML tables are often used in web pages merely for formatting.

hospital where Aldrin was born straddles both Glen Ridge and Montclair). A system might also choose to discard one triple (because, in this case, place of birth is a functional relation, that is, for each subject there can be only one “true” object).

An excellent overview of the creation of state-of-the-art knowledge bases is given in the tutorial by Bordes and Gabrilovich [2015].

4.3.2 Systems for Relationship Extraction from Text

Early approaches to relationship extraction make use of hand-crafted rules or patterns. A classical example is the pattern *NP such as NP*, which if matched in a text likely points to a *hyponymy* relation between the two noun phrases [Hearst, 1992].

The next generation of systems was based on supervised classification using linguistic features such as phrase chunks and dependency paths [Zhou et al., 2005; Fundel, Küffner, and Zimmer, 2007] or tree kernels [Zelenko, Aone, and Richardella, 2003]. Zhou et al. [2005] report 55.5% F-measure (63.1% precision and 49.5% recall) over a set of 43 relations on a corpus of the NIST Automatic Content Extraction (ACE) program.¹⁴ Again, we refer to the survey by Sarawagi [2008] for a good overview.

A common problem for supervised approaches is that labeled training data is required for each relation to be extracted. Therefore, recent approaches make use of *distant supervision* [Mintz et al., 2009] to derive (noisy) training data. The idea is to find training examples for each relation, by finding sentences in which entities, that are known to be in the given relation, co-occur. This is possible with large-scale public domain knowledge bases like Freebase, covering many relations and entities, and a large text corpus, where mentioned entities have been identified. Note that distant supervision is a technique that can be applied for other tasks as well. In general, whenever noisy training data can be derived using an authoritative source, one can speak of distant supervision.

Of course, the assumption that co-occurring entities are in the given relation does not always hold. For example, the entities *Neil Armstrong*

¹⁴<https://www.ldc.upenn.edu/collaborations/past-projects/ace>

and *Wapakoneta* can co-occur in a sentence because it states that Armstrong was born in Wapakoneta or because he took flying lessons there. Hence, recent approaches focus on better learning from this kind of noisy data [Riedel, Yao, and McCallum, 2010; Hoffmann et al., 2011; Surdeanu et al., 2012].

There is no standard annual benchmark for evaluation, and results differ based on the considered relations and used corpus. The much compared-to work by Hoffmann et al. [2011] reports around 60% F-measure (72.4% precision and 51.9% recall) across all extracted relations.

The first approaches to Open Information Extraction (OIE) mainly used patterns over shallow NLP, e.g., part-of-speech tags. Given a set of seed entities that are in a specified relation, TextRunner learns surface patterns from a text corpus and the initial bootstrap set is enriched with additional entities [Yates et al., 2007]. Later systems combined manually crafted rules with classifiers learned via this bootstrapping process, e.g., ReVerb [Fader, Soderland, and Etzioni, 2011; Etzioni et al., 2011].

More recent systems tend to utilize deeper NLP (dependency or constituent parses), e.g., OLLIE [Mausam et al., 2012] learns patterns on dependency parses. The currently best approaches use manually crafted rules over deep NLP, notably ClausIE [Corro and Gemulla, 2013] and CSD-IE [Bast and Haussmann, 2013] (extended by Bast and Haussmann [2014] to make triples more informative). Both systems report around 70% of correct extractions with around twice as many correct extractions as OLLIE [Mausam et al., 2012].

Triples from OIE systems can be used for semantic search in a variety of ways. In [Fader, Zettlemoyer, and Etzioni, 2013], the triples are searched directly, with parts of the query being matched to parts of the triples, making extensive use of paraphrases. This approach only works when the elements from the result sets correspond to individual triples. A demo of this kind of search is provided under <http://openie.allenai.org>. In [Bast et al., 2014b], triples are used to establish semantic context (which entities and words “belong together”) in semi-structured search on combined data; the system is

described in Section 4.6.2. In principle, OIE triples could also be used for knowledge base construction. However, all of the systems described in Section 4.3.1 below work with a fixed set of relations. This takes away the burden of the problem of predicate name resolution (which is hard, see Section 4.8.1). Additionally, the schema of the knowledge base provides a filter on which triples are actually useful. For example, OIE systems also extract triples like

John cheered for his team

that are usually not desirable to include in a knowledge base.

4.3.3 Systems for Specialized Extraction

WebKB [Craven et al., 1998] was one of the first systems to extract triples from hyperlinked documents, namely the website of a computer science department. In their approach, web pages stand for entities (for example, the homepage of a person stands for that person) and links between web pages indicate relations (for example, a link between a person's homepage and the department homepage is a strong indicator that that person works in that department). The correspondence between web pages and entities is learned in a supervised fashion using a Naive Bayes classifier with standard word features. Relations are also learned using FOIL (rule-based, supervised learning) with link paths (for example, a link from a person to a department) and anchor text (for example, the word *department* in the anchor text) as features. In their evaluation, 450 instances of 6 classes are classified with 73% precision and 291 instances of 3 relations are extracted with 81% precision.

EXALG [Arasu and Garcia-Molina, 2003] is a system for gathering knowledge from websites that fill templates with structured data. The goal is to deduce the template without any human input, and then use the deduced template to extract data. Mapping the extracted data to an existing ontology is not part of the system. Technically, the system works in two stages. In the first stage, it collects tokens that occur (exactly) equally often and thus indicate a template (e.g., a label like *Name:*). In the second stage, the data values are extracted. These are expected to be found between the re-occurring tokens from stage one.

Limaye, Sarawagi, and Chakrabarti [2010] present a system that extracts structured information from tables contained in web documents. In a preprocessing step, for each table, its cells, columns, and column pairs are mapped to entities, types, and relations from the YAGO knowledge base. For example, consider a table with two columns: names of persons and their birth place. The cells are mapped to particular persons and places, respectively, the columns are mapped to the types *person* and *location*, and the column pair is mapped to the relation *born in*. The mappings are learned using features such as: the similarity between the entity name and the text in the cell, the similarity between a relation name and a column header, and whether entity pairs from a labeled relation are already in this relation according to YAGO. WebTables [Cafarella et al., 2008] is a system for finding web tables in the first place. For example, given the keyword query *city population*, find tables on the Web containing information about cities and their population.¹⁵

4.3.4 Systems for Knowledge Base Construction

YAGO [2007] is a knowledge base originally obtained from Wikipedia’s infoboxes and from linking Wikipedia’s rich category information to the WordNet [Miller, 1992] taxonomy using basic NLP techniques. For example, the Wikipedia category *German Computer Scientists* can be (easily) linked to the WordNet category *Computer Scientist*, and from the WordNet taxonomy one can then infer that an entity with that category is also a *Scientist* and a *Person*. More recent versions of YAGO also contain statements from matching patterns in text, as well as extensive spatial and temporal information [Hoffart et al., 2013].

DBpedia [Auer et al., 2007] is a community effort to extract structured information from Wikipedia. The most important part are templates that extract structured data from Wikipedia infoboxes. However, there are also other extractors, including some that harvest information from full text using NLP techniques [Lehmann et al., 2015]. For example, there is an extractor that infers the gender of a person from the usage of pronouns in the person’s article.

¹⁵WebTables is used in several Google products; see [Balakrishnan et al., 2015].

The Never-Ending Language Learner (NELL) [Carlson et al., 2010; Mitchell et al., 2015] is a system that constructs a knowledge base from the Web in a staged fashion, where previously learned knowledge enables further learning. NELL has been running 24 hours/day since January 2010, and so far has acquired a knowledge base with over 80 million confidence-weighted statements. It started with a small knowledge base that defines a basic ontology (that is, a set of types and predicates of interest) and a handful of seed examples. In each cycle, the current knowledge base is used to train several components, which are then used to update the knowledge base. These components include: relationship extraction (see Section 4.3.2), removing mutually exclusive statements (see Section 4.3.1), and inference modules that generate new statements (if two people have the same parents, they should also be in a sibling relationship).

Google’s Knowledge Vault [Dong et al., 2014] is a web-scale probabilistic knowledge base that combines extractions from web content with knowledge derived from existing repositories. Knowledge Vault contains three major components. First, triple extractors that utilize distant supervision using basic NLP features derived from POS tagging, NER+NED, dependency parsing, and co-reference resolution (see Section 3). Second, graph-based priors that predict possibly missing triples (with a probability) based on what is already stored in the knowledge base. For example, one can infer a missing instance of a *sibling* relation, if two persons have the same *parent*. Missing *parent* triples can also be hinted at by a *sibling* triple, but with less confidence as the other way round. These predictions are made without manually specified rules. The final component is knowledge fusion that computes the probability of a triple being true, based on agreement between different extractors and priors. According to [Bordes and Gabrilovich, 2015], Knowledge Vault contained 302M high-confidence facts in 2015.

DeepDive [Zhang, 2015; Wu et al., 2015] provides the basic building blocks for knowledge base construction systems. An initial knowledge base and a text corpus are required. Users of DeepDive have to provide extractors, training examples, and rules. Extractors can be off-the-shelf tools or tailor-made and extract entity occurrences from the text (as

offsets) and whatever else might be a useful feature: POS tags, dependency parses, etc. Training examples are typically obtained using *distant supervision* (explained in Section 4.3.2), but can also be provided manually. Rules can state something like “if a person smokes, the person is likely to have cancer”. DeepDive then learns weights for those rules and performs inference without the developer having to worry about the algorithmic intricacies. Therefore, it creates a probabilistic model and jointly learns: (1) optimal weights for the user-defined rules, and (2) probabilities for candidate triples to be added to the knowledge base. In the example above, smoking makes cancer more probable, but that does not mean every smoker necessarily has cancer. The weight for that rule is learned from existing data and, together with evidence from the text (typical patterns or formulations), determines the confidence of new statements that might be added to the knowledge base.

Angeli et al. [2014] present a system based on DeepDive. This system was the best performing system at the 2014 TAC-KBP slot filling task [Surdeanu and Ji, 2014], which is described below. Distant supervision is performed with Freebase as a source of training data. To improve upon this, a manual feedback round is added to find features that are good indicators of the relation or not.

4.3.5 Benchmarks

For the basic task of relationship extraction, there is no widely agreed-upon benchmark. Rather, each of the systems described in Section 4.3.2 comes with its own benchmark (as briefly summarized above). The likely reason is the many variants of the extraction task: which relations to extract (fixed subset or open), the subjective judgment which triples are actually entailed by the text (and hence counted as correct), whether to extract triples or n -tuples, optimize for precision or for recall, etc.

Since 2009, the TAC conference series has a Knowledge Base Population (KBP) track. Overview papers from 2010 to 2014 are available via the conference’s website: [Ji et al., 2010; Ji, Grishman, and Dang, 2011; Mayfield, Artilles, and Dang, 2012; Surdeanu, 2013; Surdeanu and Ji, 2014]. Over the years, KBP has always featured two tasks that

are crucial to knowledge base construction: *Entity Linking* (see Section 3.2), and a so-called *Slot Filling* task, where missing facts about entities are retrieved and thus these *slots* are filled in a knowledge base. Since 2012, KBP also includes a *Cold Start* task, where a knowledge base is constructed from scratch and then evaluated as a whole.

The slot filling task is most relevant to this section. First of all, it evaluates the main aspect of knowledge base construction: to retrieve facts from a text corpus. Second, it is an example for searching with structured queries on text itself. The goal of the task is, given an entity (e.g., a particular person) together with the names of a number of relations (e.g., countries of residence), compute the missing objects (e.g., the countries of residence of the given person). All of them are attributes of either persons or organizations. Each query contains the name of the entity, its type (person or organization), and a link to one occurrence in the corpus of the task.

The text corpus consists of documents from multiple sources, with newswire text and web documents (1 million documents each) making up the biggest part. The knowledge base includes nodes for entities based on a dump of the English Wikipedia from October 2008. Results are evaluated against manually judged extractions based on pooling. Annotations from previous years are provided as additional training data to facilitate the use of the reference knowledge base.

The slot filling task is consistently lively with 15, 15, 11, 19, and 18 participants over the years. The best performing system in 2014 is the system by Angeli et al. [2014] described above. The system achieves 37% F-measure with 55% precision and 28% recall.

The cold start task introduced in 2012 has become its own track and replaced the classic KBP track in 2015 [Mayfield and Grishman, 2015]. Differently from the other tasks, no knowledge base is given as input. Instead, it is built from scratch using a given document collection and a predefined schema. This collection consists of 50K English documents from newswire text and discussion forum posts. According to the schema, systems have to recognize person, organization, and geopolitical entities (entity discovery task), their relations (slot filling task) in the text corpus, and populate a knowledge base.

Also in 2012, the TREC conference series introduced a Knowledge Base Acceleration (KBA) track [Frank et al., 2012; Frank et al., 2013; Frank et al., 2014]. The streaming slot filling task is similar to the slot filling task of KBP, except that the data is given as a stream (with time stamps for each document), and the knowledge about entities evolves over time. As the stream of documents progresses, the entities change and evolve, so KBA systems must detect when vital, new information appears that would motivate an update to the knowledge base. The data comes from the Stream Corpus (see Table 2.1). Systems are evaluated by similarity between their slot fills and those found by humans (using cosine similarity between word vectors when taking all slot fills as bags of words). The best system [Qi et al., 2014] achieves a similarity of 61%. It makes use of training data from the non-streaming TAC KBP task (described above) to learn patterns on dependency parses of sentences (see Section 3.3).

4.4 Keyword Search on Knowledge Bases

Data	A knowledge base, as described in Section 2.1.2
Search	Keyword search, as described in Section 2.2.1 The query is a sequence of (typically few) keywords; the result is a SPARQL query or a ranked list of matching items from the knowledge base
Approach	Match keywords to entities from the knowledge base; generate candidates for SPARQL queries from these matching entities; rank candidate queries using graph, lexical, and IR measures; some overlap with the techniques from Section 4.8
Strength	Easier access to structured data for simple queries
Limitation	Is bound to fail for complex search intents that cannot be adequately (unambiguously) expressed by a keyword query

The main strength of a knowledge base is that even complex queries can be asked with precise semantics. The main drawback is that it is challenging to formulate these queries. For arbitrarily complex search requests, a complex query language is inevitable. However, for relatively simple search requests, a simpler kind of search is feasible. Keyword search has the strong benefit of being an established search paradigm that users are already accustomed to.

There is a small overlap with Section 4.8 on *Question Answering on Knowledge Bases*. According to our discussion at the beginning of Section 2.2, we distinguish systems by technique and not by the apparent form of the query. The core of the approaches in this section is to match keywords to entities and then find small subgraphs in the knowledge base connecting these entities. The approaches in Section 4.8 go further, for example, by also trying to match relation names, or by considering grammatical structure.

4.4.1 Basic Techniques

Systems for keyword search on knowledge bases, or more generally on relational databases, view the data as a graph. For knowledge bases, the graph structure is already given, for relational databases, it is induced, e.g., by foreign key relations. Keywords are then mapped to nodes in this graph. Typically, an inverted index over the (words of the) entity, class, or relation names is used. This allows to efficiently match keywords to nodes of the graph during run-time. Using standard techniques as described in Section 4.1, word variants and synonyms can be matched as well; for example, matching the keyword *cmu* to a knowledge base entity *Carnegie Mellon University*.

A problem is that keyword queries might mention relations differently from how they are represented in a knowledge base or might not mention them at all. For example, the keyword query *films by francis ford coppola* doesn't explicitly mention a *directed* relation. Therefore, systems try to connect the elements that were identified via the keywords, to form a connected (sub-)graph. This can be done by exploring the neighborhood of identified elements and finding the smallest (spanning) tree connecting all elements. Often, this is an instance of the Steiner tree problem, which is NP-complete. Hence, a lot of work tries to find efficient and good approximations. From the matched graph, a structured query can be derived, for example, by replacing identified classes with result variables.

Because words from the query can match several components of the graph, the translation results in several candidate queries which need to be ranked. Techniques for ranking these make use of two main factors: the relevance and the structure of the matching (sub-)graph. For relevance, ranking functions from information retrieval (see Section 4.1 on *Keyword Search on Text*) can be adapted to this setting, e.g., by assuming that each matched subgraph corresponds to a virtual document. In addition, the popularity of matching nodes (for example, derived via PageRank) and the quality of the keyword mappings (e.g., via the Levenshtein distance) can be considered. The structure of the matching graphs is incorporated, for example, by ranking smaller graphs higher. The intuition behind this is that simpler queries are more likely to be

correct. Similarly, the number of joins of the corresponding query can be considered (as a proxy for query complexity).

To improve usability, some systems also include user feedback in the translation process. This is done, for example, by suggesting keyword completions that lead to results, or by allowing the user to select the correct interpretation for each keyword (when several are possible).

Below, we first introduce systems designed for keyword queries on general relational databases, followed by systems specifically designed for knowledge bases.

4.4.2 Systems for Keyword Search on Relational Databases

Keyword search on relational databases is an actively researched field on its own. The survey by Yu, Qin, and Chang [2010] gives a good overview. Coffman and Weaver [2010] and Coffman and Weaver [2014] perform a qualitative evaluation of many state-of-the-art systems on a benchmark they introduce for that purpose (see below).

DBXplorer [Agrawal, Chaudhuri, and Das, 2002], DISCOVER [Hristidis and Papakonstantinou, 2002] and BANKS [Bhalotia et al., 2002] were the first prominent systems for keyword search on relational databases. DBXplorer and DISCOVER use the number of joins to rank answers, while BANKS tries to find the smallest matching subgraph. Subsequent work refines and combines the techniques mentioned above to improve results.

Tastier [Li et al., 2009] includes the user in answering keyword queries. In addition to translating to a SQL query it provides context-sensitive auto-completion of keyword queries, similar to what is described in [Bast and Weber, 2006]. This is achieved via specialized data structures (mainly a trie over words in the database) that allow computing completions of keywords that lead to results.

GraphLM [Mass and Sagiv, 2012] applies language models for ranking. Keyword queries are matched to subgraphs which correspond to a possible answer. Nodes in each subgraph have text associated with them via different fields: title (e.g., names), content (all attributes and their values) and structural (only attribute names). This allows learning a language model for each subgraph and field, which can then be

used to compute, e.g., $p(q|a_{title})$, the probability that a query q is generated by the title field of subgraph a . Ranking also incorporates node and edge weights. Intuitively, nodes with high in-degrees and unique edges are more important. The system outperforms all of the previous systems on the benchmark by Coffman and Weaver [2010] that is described in Section 4.4.2 below.

4.4.3 Systems for Keyword Search on Knowledge Bases

SemSearch [Lei, Uren, and Motta, 2006] was one of the first systems for keyword queries on knowledge bases. It accepts keyword queries with some additional structure, e.g., there is syntactic sugar for including *types* in queries and operators AND and OR are supported. An inverted index is used that maps keywords to classes, instances and properties of the knowledge base. The matching elements are combined in all possible ways using several query templates to obtain a structured query in SeRQL (a predecessor of SPARQL).

Tran et al. [2007] suggest a similar method to translate keyword queries to SPARQL queries. Keywords are mapped to entities (via their URIs and labels) of the knowledge base via an inverted index (implemented with Lucene). Starting at the matched entities, the knowledge base is explored in order to find subgraphs connecting the matched elements. The matching subgraphs are ranked by the lengths of their paths (with the intuition that smaller lengths correspond to better paths) and translated into a SPARQL query.

SPARK [Zhou et al., 2007] uses more sophisticated techniques, e.g., synonyms from WordNet and string metrics, for mapping keywords to knowledge base elements. The matched elements in the knowledge base are then connected by finding minimum spanning trees from which SPARQL queries are generated. To select the most likely SPARQL query, a probabilistic ranking model that incorporates the quality of the mapping and the structure of the query is proposed.

Zenz et al. [2009] follow an interactive and incremental approach to translate a keyword query into a SPARQL query. For each keyword provided by the user, a choice of a possible interpretation (with respect to the final SPARQL query) is presented. When the user selects an

interpretation for one keyword, the number of possible interpretations of the remaining keywords is reduced. This allows to incrementally construct complex SPARQL queries from keyword queries.

Hermes [Tran, Wang, and Haase, 2009] can search on multiple, possibly interlinked, knowledge bases.¹⁶ In a preprocessing step, the knowledge bases are partly unified using maps between the various elements of the knowledge bases and their respective ontologies. Hermes also precomputes a map from potential search terms to elements of the knowledge bases. Keyword queries can then be mapped to candidate subgraphs in the resulting meta knowledge base. The candidates are ranked, preferring shorter paths containing important elements which match the keywords well. The system is evaluated on a combination of seven knowledge bases (including DBpedia, Freebase, and GeoNames, see Table 2.2) with a total of around 1.1B triples.

Pound et al. [2012] focus on keyword queries from the logs of the Yahoo web search engine. Keywords of a query are first tagged as entity, type, or relation mentions. The mentions are then arranged by mapping them to one of ten structured query templates. Both steps are learned via manually annotated queries from a query log using straightforward machine learning. In a final step, standard techniques as described above are used to map the mentions to entities and relations of a knowledge base. The system is evaluated using 156 manually annotated queries from the Yahoo query log and YAGO as a knowledge base.

4.4.4 Benchmarks

Coffman and Weaver [2010] introduce a benchmark on three datasets: selected data from Mondial¹⁷ (geographical knowledge), IMDB, and Wikipedia, respectively. For each dataset, 50 queries were manually selected and binary relevance judgments for results are provided by identifying all possible correct answers. Evaluation metrics are those

¹⁶As such, it seems that Hermes should be described in Section 4.5 on *Keyword Search in Combined Data*. However, due to the unification process, the system is technically more similar to the systems in this section.

¹⁷<http://www.dbis.informatik.uni-goettingen.de/Mondial>

typical for information retrieval: precision at 1, mean reciprocal rank, and mean average precision. The authors evaluate nine recent state-of-the-art system on this benchmark. Many previous claims cannot be corroborated, which shows the shortcomings of previous evaluations. The evaluation also shows no clear winner, but that most systems score comparably on average, with different systems performing best on different datasets. In a follow-up evaluation, the GraphLM system [Mass and Sagiv, 2012] discussed above produced the consistently best results.

Balog and Neumayer [2013] assembled queries from a variety of previous benchmarks by mapping relevant entities to DBpedia. Some of these benchmarks were originally designed for semantic web data (like BTC; see Table 2.3), but the best systems mostly return results from the (comparably tiny) DBpedia part only. The new benchmark includes keyword queries (e.g., from the TREC Entity Tracks; see Section 4.5.3) as well as natural language queries (e.g., from QALD-2; see Section 4.8.5). Evaluation metrics are MAP (mean average precision) and precision at 10. Several baselines have been evaluated on the benchmark but adoption is slow. The current best performing system is from Zhiltsov, Kotov, and Nikolaev [2015], which achieves a MAP of 23%, an absolute improvement of 4% over one of the baselines.

In theory, benchmarks for question answering on knowledge bases (discussed in Section 4.8.5) could also be used by transforming natural language queries into keywords. In fact, some of those benchmarks also provide keyword versions of the questions. Obviously, this will fail when questions are more complex than what a keyword query can reasonably express.

4.5 Keyword Search on Combined Data

Data	Combined data, as described in Section 2.1.3 Specifically here, text with entity annotations or semantic web data
Search	Keyword search, as described in Section 2.2.1 Results are ranked lists of entities, maybe augmented with text snippets matching the query; optionally restricted to entities of a given type
Approach	For each entity, create a virtual text document from (all or a selection of) text associated with it; search these documents using techniques from Section 4.1; alternatively, first search given text using techniques from Section 4.1, then extract entities from the results and rank them
Strength	Easy-to-use entity search on combined data; works well when the data provides sufficiently strong relevance signals for the keyword, just as in keyword search on text
Limitation	Similar precision problems as for keyword search on text; see the box at the beginning of Section 4.1

Many keyword queries actually ask for an entity or a list of entities instead of a list of documents. In a study by Pound, Mika, and Zaragoza [2010] on a large query log from a commercial web-search engine, 40% of queries are for a particular entity (e.g., *neil armstrong*), 12% are for a particular lists of entities (e.g., *astronauts who walked on the moon*), and 5% are asking for a particular attribute of a particular entity (e.g., *birth date neil armstrong*).

Section 4.4 discusses one option for such queries: keyword search on knowledge bases. In this section, we consider combined data, which for keyword search is typically either semantic web data (multiple knowledge bases with different naming schemes and extensive use of string

literals; see Section 2.1.3) or text with entity annotations (this is the simplest form of text linked to a knowledge base; also see Section 2.1.3).

4.5.1 Basic Techniques

There are two prevalent approaches: search in virtual documents (one per entity) and standard keyword search on text followed by an entity extraction and ranking step.

In the *virtual document* approach, all or some of the data related to a particular entity (relation names, object names, string literals) is collected in a single virtual document for that entity. This makes particular sense for semantic web data, where the extreme heterogeneity of the data makes a structured search hard. Also, in some applications, there is a document per entity in the first place. A notable example is Wikipedia, which is used in all of the INEX benchmarks, discussed in Section 4.5.3 below. Given one document per entity (virtual or real), the result corpus can be searched using techniques from Section 4.1. The ranking of this kind of documents is discussed in detail in Section 5.1.1. Efficient indexing is discussed in Section 5.2.1. All of the systems described in Section 4.5.2 below are based on the virtual document approach, and they are all for semantic web data.

In the *search and extract* approach, the first step is keyword search on text. Many systems use one of the off-the-shelf systems from Section 4.1.2 for this task, or a web search engine like Google. In a second step, entities are extracted from the results (either from the full documents or only from the result snippets). This is trivial for collections like FACC (see Table 2.3), where entity annotations are part of the data. In a third step, entities are ranked. This is where the intelligence of systems using this approach lies. We hence describe them in Section 5.1.2 from our section on *Ranking*.

In both of these approaches, entity resolution (that is, different names or URIs for the same entity) is a challenge. The Semantic Web allows users to provide explicit links between such entities, notably via relations such as *owl:sameAs* or *dbpedia:redirect/disambiguate*. Not surprisingly, making use of such links can considerably improve result quality [Tonon, Demartini, and Cudré-Mauroux, 2012]. Section 5.1.1

describes a method that uses language models, which are normally used for ranking, for automatically establishing *owl:sameAs* links in semantic web data. Section 5.1.3 is about ranking interlinked entities (obtained from a semantic web search) in general, where *owl:sameAs* links also influence scores.

A special case of keyword search on combined data is *expertise retrieval*, where the goal is to retrieve a list of experts on a given topic. For example, find experts on *ontology merging* from a collection of W3C documents. The experts are persons, and it is either part of the problem to identify their mentions in the text (this is an instance of NER+NED; see Section 3.2) or these annotation are already provided. Note that the underlying knowledge base is then a simplistic one: just the entities (persons) and their names. The typical approaches are via virtual documents or via search and extract, as discussed above. A recent survey is provided by Balog et al. [2012].

4.5.2 Systems (all for Semantic Web Data)

Guha, McCool, and Miller [2003] describe an early prototype for search on the Semantic Web. At that time, hardly any semantic web data was available yet. The data was therefore artificially created via scraping¹⁸ from a small selection of websites. Their main use case is single-entity search, that is, part or all of the query denotes an entity. Aspects discussed are disambiguation of entity names in the query (user interaction is suggested as a solution), disambiguation of entity names in matching documents (this is essentially the NER+NED problem from Section 3.2), and which of the usually many triples about the entity to show (various simple heuristics are discussed). The final result about the matching entity is shown in an infobox on the right, similar to how the large web search engines do it nowadays (except that those infoboxes do not come from the Semantic Web, but rather from a single, well-curated knowledge base).

Swoogle [Ding et al., 2004] was one of the first engines to provide keyword search on the Semantic Web. Swoogle indexes *n*-grams to leverage

¹⁸Scraping refers to extracting structured data from ordinary websites, often via simple web-site specific scripts.

the information hidden in the often long URIs of entity and relation names. Also, an n -gram index enables approximate search. The index is augmented by metadata, so that search results can be restricted by certain criteria (e.g., to results in a particular language). The system also comprises a crawler and custom ranking function. As of this writing, there was still a demo available at <http://swoogle.umbc.edu>.

Falcons [Cheng, Ge, and Qu, 2008] provides a similar functionality as Swoogle, with the following additional features. The search can be restricted to entities of a given certain type (e.g., to type *conference* when the query¹⁹ is *beijing 2008*). The search can also be restricted to a particular knowledge base (e.g., to only DBpedia). In the (default) entity-centric view, matching triples are grouped by entity, and for each entity a selection of the matching triples are displayed. Different URIs from the same entity are not merged. As of this writing, there was still a demo available at <http://ws.nju.edu.cn/falcons>.

Sindice [Oren et al., 2008] offers similar functionality on a distributed very large scale by using Hadoop and MapReduce. It also inspects schemata to identify properties that uniquely identify an entity, e.g., *foaf:personalHomepage*, which allows retrieval based on the property and its value. The system is not designed to be an end-user application but to serve other applications that want to locate information sources via an API. Unfortunately, as of this writing, the service was no longer available.

Glimmer [Blanco, Mika, and Vigna, 2011] constructs a virtual document for each entity using fielded BM25F. The particular index is described in Section 5.2.1. This allows customizing the contribution weight of contents from certain data sources and relations. Both quality and performance are evaluated on the WDC dataset (see Table 2.3) with queries from the SemSearch Challenge 2010 (see Section 4.5.3 below). Queries are keywords, possibly annotated by fields or relations they should match. As of this writing, a live demo is available under <http://glimmer.research.yahoo.com>.

¹⁹The WWW'08 conference, where this paper was presented, took place in Beijing.

As of this writing, there is no single system that searches the totality of semantic web data with a coverage and result quality even remotely comparable to that of the large commercial web search engines. This is largely due to the fact that, although the data is large in size, the amount of information contained is tiny compared to the regular web. It is also noteworthy that approaches with good results, like Glimmer above, boost high-quality contents like DBpedia. Indeed, as of this writing, all major commercial systems rely on internal well-curated knowledge bases; see Section 4.8 on *Question Answering on Knowledge Bases*.

4.5.3 Benchmarks

There are three notable series of benchmarks for keyword search on combined data, in particular, semantic web data: the TREC Entity Track (2009 - 2011), the SemSearch Challenge (2010 and 2011), and the INEX series of benchmarks (2006 - 2014). The QALD (Question Answering on Linked Data) benchmarks are described in Sections 4.8.5 (Question Answering on Knowledge Bases) and 4.9.4 (Question Answering on Combined Data).

We remark that participation in these competitions was low (generally below 10 participating groups, sometimes only a couple of participants). However, the datasets and queries continue to be used in research papers related to semantic search.

TREC Entity Track (2009 - 2011): An overview of each of the three tracks is provided in [Balog et al., 2009; Balog, Serdyukov, and Vries, 2010; Balog, Serdyukov, and Vries, 2011]. A typical query is:

airlines that currently use boeing 747 planes

The central entity of the query (*boeing 747*) and the type of the target entities (*airlines*) was explicitly given as part of the query. There were two kinds of datasets: text (ClueWeb, see Table 2.1) and semantic web data (BTC'10, see Table 2.3).²⁰

²⁰In the TREC Entity Track 2009, only ClueWeb'09 was used. In the TREC Entity Track 2011, the Sindice dataset was used instead of BTC'10. However, the Sindice dataset is no longer available, which is why we do not list it in Table 2.3.

The best systems that worked with the BTC'10 dataset used the virtual document approach described in Section 4.5.1 above. That is, although the queries appear more as natural language queries (see the example above), the processing is clearly keyword search style. According to our discussion at the beginning of Section 2, we make the distinction between these two kinds of search by technique. This also explains why we describe this benchmark in this section and not in Section 4.9 on *Question Answering on Combined Data*.

The best system that worked with the ClueWeb'09 dataset used the extract and search approach described in Section 4.5.1 above. It is mainly about ranking, and hence described in Section 5.1.2 (right at the beginning). Interestingly, the system chose to ignore the official dataset and instead used Google Search for the initial retrieval step.

The best results for the main task (related entity finding, like for the query above) were an nDCG@R of 31%, 37%, and 25% in 2009, 2010, and 2011, respectively. The best result for the related task of entity list completion (where some result entities are given) was a mean average precision of 26% in 2010.

SemSearch Challenge (2010 and 2011): An overview over each of these two challenges is provided in [Halpin et al., 2010] and [Blanco et al., 2011]. In 2010, queries were keyword queries asking for a single entity (for example, *university of north dakota*). In 2011, there were two tasks: keyword queries for a single entity (like in 2010, but new queries) and keyword queries for a list of entities (for example, *astronauts who landed on the moon*). Both challenges used the BTC'09 dataset (see Table 2.3).

The best approaches again construct virtual documents and use a fielded index and corresponding ranking function. The winner in 2010 achieved a precision at 10 of 49% and a mean average precision of 19%. In 2011, the best result for the single-entity task was a precision at 10 of 26% and a mean average precision of 23%. The best result for the entity-list task was a precision at 10 of 35% and a mean average precision of 28%.

INEX (2006 - 2014): INEX (Initiative for the Evaluation of XML Retrieval) has featured several ad-hoc search tasks. The dataset was

Wikipedia with an increasing amount of annotations, all represented in XML; see Section 2.1.3 for an example. From 2006 - 2008, annotations were obtained from Wikipedia markup (in particular: infoboxes, links, and lists). In 2009, cross-references to entities from YAGO (see Table 2.2) were added to the Wikipedia links, as well as for each page as a whole. In 2012, additional cross-references to DBpedia (see Table 2.2) were added. The resulting dataset is Wikipedia LOD (see Table 2.3).

The goal of the early ad-hoc tasks (2006 - 2010) was similar to that of the TREC ad-hoc tasks described in Section 4.1.3. Queries were also similar, for example, *olive oil health benefit* (from 2013) or *guitar classical bach* (from 2012). One notable difference was the focus on the retrieval of (XML) elements rather than whole documents, and the incorporation of the proper focus of these elements (not too large and not too small) in the quality measure. See [Gövert et al., 2006] for an overview paper on this aspect of XML retrieval.

The INEX Entity Ranking Track (2007 - 2009) is similar to the TREC Entity Track from above: given a keyword query (describing a topic) and a category, find entities from that category relevant for that topic. For example, find entities from the category *art museums and galleries* that are relevant for *impressionist art in the netherlands* (from 2007).

The INEX Linked-Data Track (2012 and 2013) explicitly encouraged the use of the external knowledge bases (YAGO and DBpedia) to which the Wikipedia content was linked. However, few participants made use of that information and the results were inconclusive.

Our take on the usability of XML-style retrieval for semantic search is as follows. XML shines for deeply nested structures, with a mix between structured and unstructured elements. Indeed, query languages like XPath and XQuery are designed for precise retrieval involving complex paths in these structures. However, datasets actively used in semantic search at the time of this writing have a flat structure (triples or simple links from the text to entities from the knowledge base; see Tables 2.2 and 2.3). The core challenge lies in the enormous size and ambiguity of the data (queries, text, and entity and relation names), which is nothing where XML can specifically help.

4.6 Semi-Structured Search on Combined Data

Data	Combined data, as described in Section 2.1.3 Specifically here, text linked to a knowledge base
Search	Structured search, as described in Section 2.2.2, extended with a keyword search component Results are ranked lists of entities, maybe augmented with matching text snippets or matching information from the knowledge base
Approach	Store data in an inverted index or extensions of it; use separate indexes for the text and the knowledge base or use tailor-made combined indexes; provide special-purpose user interfaces adapted for the particular kind of search
Strength	Combines the advantages of text (widely available) and knowledge bases (precise semantics); good for expert search and as a back end for question answering
Limitation	Queries with a complex structured part have the same usability problems as described at the beginning of Section 4.2

Since combined data contains both structured and unstructured elements, it is natural that queries also contain a mix of structured and unstructured elements. Simple text search extensions of SPARQL are discussed already in Section 4.2. This section considers more sophisticated extensions.

4.6.1 Basic Techniques

Text linked to a knowledge base allows searches for co-occurrences of arbitrary keywords with arbitrary subsets of entities, as specified by a structured query on the knowledge base. A simple example would be to search for all *politicians* (all entities in the knowledge base with that

profession) that co-occur with the keywords *audience pope*. This could be expressed as an extended keyword query, where some keywords are concepts from the knowledge base, for example:

type:politician audience pope

This kind of search can easily be supported by an inverted index, with an artificial index item (like the *type:politician*) added for each mention of a politician in the text. Alternatively, XML search engines supporting languages like XPath or even XQuery could be used. However, this would be cracking a nut with a sledgehammer; see the discussion at the end of Section 4.5.3, after the description of the INEX benchmarks.

As a more complex example, consider the example query from the introduction *female computer scientists who work on semantic search*. This is naturally expressed as a structured query (that expresses the knowledge base part) extended with a keyword search component (that expresses the co-occurrence with the given keywords). In the syntax of the Broccoli system, discussed below, this can be written as:

```
SELECT ?p WHERE {
  ?p has-profession Computer_Scientist .
  ?p has-gender Female .
  ?p occurs-with "semantic search" }
```

For this more general class of queries the simple annotation trick fails, at least for a knowledge base of significant size. We then cannot annotate each entity in the text with all the information that is available about it in the knowledge base. The ways to index such data, as well as their strengths and limitations, are discussed in detail in Section 5.2.2 on *Semi-Structured Search Based on an Inverted Index*.

It is important to understand the difference between a relation like *occurs-with* and a simple text search extension like *bif:contains* discussed in Section 4.2.1. Consider the query above with the last triple replaced by

```
?p has-description ?d . ?d bif:contains "semantic AND search"
```

That query requires each matching entity to stand in a *has-description* relation to a string literal containing the desired keywords. This is

unlikely to be fulfilled by a typical knowledge base. In contrast, the original query from above only requires that a matching entity co-occurs with the given keywords somewhere in the text corpus. This is realistic for a sufficiently large text corpus.

4.6.2 Systems

KIM [Popov et al., 2004] was one of the first systems to provide semi-structured search on text linked to a knowledge base, as described above. Results are documents that mention entities from the structured part of the query as well as the specified keywords. The text is indexed with Lucene (see Section 4.1.2), including for each entity an inverted index of the occurrences of that entity in the text. The knowledge base is indexed with Sesame (see Section 4.2.2). The results from the two indexes are combined by computing the union of the inverted lists of the entities matching the structured part of the query. This runs into efficiency problems when the structured part matches very many entities (for example, a structured query for just *person*).

Ester [Bast et al., 2007] provides similar functionality as KIM, but achieves scalability with a special-purpose combined index, adapted from [Bast and Weber, 2006]. The index also provides fast query suggestions after each keystroke, for words from the text as well as for elements from the knowledge base. The system was evaluated on a variant of the Wikipedia LOD dataset (see Table 2.3).

Broccoli [Bast et al., 2012; Bast et al., 2014b] provides extended keyword search as well as extended structured search; an example query for the latter is given above. The structured part of the query is restricted to tree-like SPARQL queries. Co-occurrence of entities from the text with entities from the knowledge base can be restricted to the semantic contexts from [Bast and Haussmann, 2013], as explained in Section 4.3.1 on *Relationship Extraction from Natural Language Text*. Interactive query suggestions are provided, and an elaborate user interface is provided. Results can be grouped by entity, with matching text snippets. A tailor-made index for the efficient support of these features is provided, which is explained in Section 5.2.2. The system is evaluated on a variant of the Wikipedia LOD dataset (see Table 2.3) with

queries adapted from the TREC Entity Track 2011 and the SemSearch Challenge 2011, as explained in Section 4.6.3 below. As of this writing, a live demo is available: <http://broccoli.cs.uni-freiburg.de>.

Mimir [Tablan et al., 2015] is an extension of KIM. Compared to KIM, simple queries are implemented more efficiently (for example, a search for *cities* that occur with certain keywords), and full SPARQL is supported for the structured part of the query (though not particularly efficiently when combined with keyword search). For the text corpus, MG4J is used (see Section 4.1.2). The ranking function is customizable, in particular, BM25 is supported (see Section 4.1.1). Results are matching documents, grouping by entities is not supported. The software is open source.

4.6.3 Benchmarks

There are no widely used benchmarks that are explicitly designed for semi-structured search on combined data.

However, the benchmarks from Section 4.5.3 (TREC Entity Track, SemSearch Challenge, and INEX) can be easily adapted for this scenario. Namely, most of the queries of these benchmarks have a part pertaining to information best found in a knowledge base and a part pertaining to information best found in text. For example, for the query *astronauts who landed on the moon* (SemSearch Challenge 2011, entity-list task), the information who is an astronaut is best taken from a knowledge base, whereas the information who landed on the moon is best found in text. The semi-structured representation for this query is similar to the example given in Section 4.6.1 above.

The Broccoli system, discussed in Section 4.6.2 above, has adapted the queries from the TREC Entity Track 2009 (main task: related entity finding) and the SemSearch Challenge 2011 (entity-list task) in this manner. On a variant of the Wikipedia LOD dataset (Table 2.3), an nDCG of 48% and 55%, respectively, is achieved.

The queries for the QALD (Question Answering on Linked Data) benchmarks, which are described in Sections 4.8.5 and 4.9.4, can be adapted in a similar way. QALD-5 features a track with explicitly semi-structured queries; see Section 4.9.4.

4.7 Question Answering on Text

Data	Text documents, as described in Section 2.1.1
Search	Natural language queries, as described in Section 2.2.3; the results are passages or statements from the text that answer the question
Approach	Derive suitable keyword queries and the answer type from the question; extract answer candidates from the (many) result snippets and rank them; optionally use reasoning and an external general-purpose knowledge base
Strength	The most natural kind of queries on the most abundant kind of data
Limitation	Questions that require combination of facts not found in the text; or questions with complex structure

Question answering on text became popular in the early 1990s, when large amounts of natural language texts started to become available online. With the advent of the world wide web, the field blossomed. According to the scope of this survey, as explained in Section 1.2, we here focus on so-called *extractive* question answering, where the desired answers can be found in the text and no synthesis of new information is required. Indeed, most research on question answering on text is of exactly this kind.

4.7.1 Basic Techniques

Prager [2006] gives an excellent survey of the development of the field until 2006. The survey by Kolomiyets and Moens [2011] focuses on techniques (and less on complete systems) and surveys some more recent research like co-reference resolution and semantic role labeling (as discussed in Section 3.3).

In 2007, the popular series of *TREC Question Answering* benchmarks (described below) ended. In all issues, a single system, Lymba's PowerAnswer and its predecessors, beat the competing systems by a large margin. We briefly describe that system below. At the time of this writing, we still consider it the state of the art in (extractive) question answering on text.

The field has since moved away from only text as a data source. Just around the time of the last TREC QA benchmark, large general-purpose knowledge bases like YAGO, DBpedia, and Freebase (see Table 2.1.2) started to gain momentum and comprehensiveness. This spawned extensive research activity on question answering on such knowledge bases, which we describe in Section 4.8 on *Question Answering on Knowledge Bases*. Note that a question like:

what is the average gdp of countries with a literacy rate below 50%

is relatively easy to answer from a knowledge base, but very hard to answer from text alone (unless the text contains that piece of information explicitly, which is unlikely).

At about the same time, IBM started its work on *Watson*, aimed at competing against human experts in the Jeopardy! game. Watson draws on multiple data sources, including text as well as the knowledge bases just mentioned. Therefore, we describe that work in Section 4.9 on *Question Answering on Combined Data*.²¹

Search engines like WolframAlpha or Google also accept natural language queries, but as of this writing, the answers do not come from text, but rather from an internal (well-curated) knowledge base; see Subsection 4.8.4 of the section on *Question Answering on Knowledge Bases*.

4.7.2 The START System

START [Katz, 1997; Katz, Borchardt, and Felshin, 2006] was the first web-based question answering system. It is one of the few systems with

²¹John Prager, the author of the above-mentioned survey, was a member of the team working on Watson.

a reliable online demo²², which has been up and running continuously since 1993 to this day. It answers natural language queries by first extracting structured information (basically: nested subject-predicate-object triples) from sentences and storing them in a knowledge base. Compared to full-fledged knowledge base construction, as described in Section 4.3 on *Structured Data Extraction from Text*, the constructed knowledge base does not have a single consistent schema and is fuzzy. The system answers questions by transforming them into the same triple-like representation and matching them against the knowledge base. Matched facts are then translated back to a natural language sentence that is presented to the user.

4.7.3 The PowerAnswer System

We briefly describe Lymba’s PowerAnswer [Moldovan, Clark, and Bowden, 2007], the undisputed winner of the TREC Question Answering track. The system can still be considered state of the art at the time of this writing. In particular, its basic architecture is typical for a system that does question answering on text.

Depending on the type of question (factoid, list, definition etc.), the system implements different strategies. Each strategy has the following main components:

Answer type extraction: determine the answer type of the query; for example, the answer for *who* ... could be a person or organization, but not a place or date.

Keyword query generation: generate one or more keyword queries, which are then issued to a text search engine, with standard techniques as described in Section 4.1.

Passage retrieval: retrieve passages from the documents matching these keyword queries that could possibly be an answer to the question.

Answer extraction: extract potential answers from the retrieved passages; rank those answers by a score that reflects the “relevance” to and the degree of “semantic match” with the question.

²²<http://start.csail.mit.edu/index.php>

The process involves subsystems solving many of the natural language processing problems discussed in Section 3. In particular, answer extraction often makes use of POS tagging, chunking, and named-entity recognition and disambiguation. Particular kinds of entities relevant for the kind of questions asked in the TREC benchmarks are *events*, *dates*, and *times*.

In the TREC benchmarks, the answer is eventually to come from the reference text corpus (typically AQUAINT, as described below). However, PowerAnswer also issued keyword queries against external sources like amazon.com, imdb.com, and Wikipedia to find candidate answers. These were then used, in turn, to find correct answers in the TREC collection.

Candidate answers are ranked using pre-trained language models and scoring functions, using state-of-the-art techniques known from keyword search on text as described in Section 4.1. PowerAnswer also makes use of COGEX, a logic prover with basic reasoning capabilities, to re-rank candidate answers. COGEX is similar to the inference engines we describe in Section 5.4 on *Inference and Reasoning* (where we restrict ourselves to publicly available systems). It generates a logic form of the question and candidate answer and performs a proof by contradiction. As part of the reasoning it also makes use of real world knowledge (e.g., that Sumatra is a part of Asia) and natural language statements (e.g., that the verb *invent* is a hyponym of *create*). The proofs (if they succeed) output a confidence score, which depends on the rules and axioms that were applied. The score is used as part of ranking the candidate answers.

4.7.4 Benchmarks

The TREC Questions Answering Track ran from 1999 - 2007, with 9 issues altogether. There is a comprehensive overview article for each year, describing the individual tasks as well as the participating systems and their results [Voorhees, 1999; Voorhees, 2000; Voorhees, 2001; Voorhees, 2002; Voorhees, 2003; Voorhees, 2004; Voorhees and Dang, 2005; Dang, Lin, and Kelly, 2006; Dang, Kelly, and Lin, 2007]. Participation was strong, with at least 20 participating groups, peaking

in 2001 with 36 groups. Lymba’s PowerAnswer, described above, and its predecessors participated and dominated the competition in each year. For example, in 2007, PowerAnswer scored an accuracy of 70.6% on factoid questions and of 47.9% on list questions with a runner-up accuracy of 49.4% and 32.4%, respectively.

All tracks made use of the AQUAINT or AQUAINT2 text corpus. The last two tracks also made use of the BLOG’06 corpus. These datasets are described in Table 2.1.

The TREC Entity Tracks (2009 - 2011) featured entity-centric search on ClueWeb (see Table 2.1) as one of their tasks. Overview papers and example queries are provided in Section 4.5 on *Keyword Search on Combined Data*, since other tasks from these tracks used semantic web data. The systems working on ClueWeb used similar techniques as described for Lymba’s PowerAnswer above. However, the Entity Track tasks additionally required that systems return an authoritative URL for each result entity, and not just its name. This made the task considerably harder.

In 2015, a new TREC *LiveQA* track was initiated, with the goal to “revive and expand the [Question Answering track described above, but] focusing on live questions from real users”. However, many of the questions asked there can hardly be considered *extractive*. One of the three examples questions from the track’s call for participation reads:

Is the ability to play an epic guitar solo attractive in a woman? Or do you see it as something aggressive and a turn off?

Apart from being sexist, such questions usually require synthesis of new information and are hence out of scope for this survey.

4.8 Question Answering on Knowledge Bases

Data	A knowledge base, as described in Section 2.1.2
Search	Natural language queries, as described in Section 2.2.3; the result is a SPARQL query or a ranked list of matching items from the knowledge base
Approach	Generate candidates for SPARQL queries by analyzing the structure of the question and mapping parts of the question to entities and relations from the knowledge base; rank query candidates and execute the top query to retrieve the answer
Strength	User-friendly access to the growing amount of data that is available in knowledge bases
Limitation	Very hard for complex queries, especially when the knowledge base is large; only a fraction of the world's information is stored in knowledge bases

Leveraging the rapidly growing amount of information in knowledge bases via natural language queries is a relatively young field. There is some overlap with Section 4.4 on *Keyword Search on Knowledge Bases*, which is discussed at the beginning of that section. The difference becomes clearer when reading and comparing Subsection 4.4.1 from that section and Subsection 4.8.1 below.

4.8.1 Basic Techniques

The goal of the typical systems from this section is the same as for the typical systems from Section 4.4: *translate* the given question to a (SPARQL) query that expresses what the question is asking for. The basic mechanism is also similar to that described in Section 4.4.1: consider a set of candidate queries (which stand for possible interpretations of the question) and from that set pick the one that represents the given question best. However, the way these candidate sets are generated and

how the best query is selected from that set is more sophisticated, going much more in the direction of what could be called “understanding” the question.

As in Section 4.4, recognizing entities from the knowledge base in the query (the NER+NED problem from Section 3.2) is a crucial component. However, all of the systems in this section also try to recognize *relation names* from the knowledge base in the question. This is harder than recognizing entities, because of the much larger variety in which relation names can be expressed in natural language.

A typical approach for recognizing relation names is via indicator words or synonyms that are learned from a text corpus by distant supervision (explained in Section 4.3.2) or by using datasets obtained via distant supervision, e.g., Patty [2013]. Another approach is to use corpora of paraphrased questions, such as Paralex [2013], to derive common paraphrases.

Natural language questions are often longer and provide more information than keyword queries. For example, compare *in what films did quentin tarantino play* to *quentin tarantino films*. The natural language question is more explicit about the expected type of result (*films*) and more precise about the relation (films in which Quentin Tarantino acted, not films which he directed). At the same time, natural language questions can also be more complex. For example, *who was born in vienna and died in berlin*.

Some of the systems below exploit this additional information by performing a linguistic analysis of the question. This is done with existing taggers and parsers (see Sections 3.1 and 3.3), or by training new special-purpose parsers. The result provides the linguistic or semantic structure of the question, which can be used to generate a template for a SPARQL query. It remains to fill in the entity and relation names. It turns out that a joint optimization of the structure (and hence the query template) and the entity and relation names works better than solving the two problems independently.

Selecting a query from the set of candidate queries is also more complex than for the systems in Section 4.4. The techniques sketched above provide a rich set of features for determining how well a candidate

query matches the given question. A typical approach is to use these features for learning to rank the candidates from given training data. This enables solving even hard questions (in the sense that the correct SPARQL query is hard to find using simple matching techniques) as long as there are enough examples in the training data. For example, answering the question *who is john garcia* with *singer* requires understanding that the *who is* part of the question is asking for the profession of the person that follows.

Section 4.8.5 below describes three widely used benchmarks: Question Answering on Linked Data (QALD), Free917, and WebQuestions. The QALD benchmarks sparked from the semantic web community, while Free917 and WebQuestions were initiated by the computational linguistic community. We first describe systems that were evaluated on QALD, followed by systems evaluated on Free917 and WebQuestions.

4.8.2 Systems Evaluated on QALD

The AutoSPARQL system [Unger et al., 2012] bases its translation on a linguistic analysis of the question. Using a lexicon of manually-designed domain-independent expressions (such as *most* or *more than*) query templates are instantiated from the structure of the question. To derive SPARQL queries, the templates are instantiated with elements from the knowledge base. Queries are then ranked by preferring prominent entities but also by considering string similarities of the knowledge base mapping. The system was evaluated on 39 of the 50 questions of QALD-1, of which it was able to answer 19 perfectly.

DEANNA [Yahya et al., 2012] formulates the task of translating a given question to a SPARQL query as an integer linear program. The program incorporates the identification of concept and relation phrases in the question, mapping these to the knowledge base, and a dependency parse to generate (SPARQL) triple candidates. Aliases from YAGO2s [2011] and relation phrases from ReVerb [Fader, Soderland, and Etzioni, 2011] are used to map to entities and relations from the knowledge base. Additionally, semantic coherence and similarity measures are incorporated. The system was evaluated on QALD-1, where it was able to answer 13 out of 27 questions correctly.

Xser [Xu, Feng, and Zhao, 2014] performs the translation in two separate phases. The first phase identifies relevant phrases (mentioned entities, relations, types) in the question, independently of the knowledge base. The identified phrases are arranged in a DAG to represent the structure of the question. Training data is used to learn a model and parser for this. The second phase maps the identified phrases to entities and relations from the knowledge base. For the experiments on DBpedia, the Wikipedia miner tool²³ is used to find matching entities, and the data from Patty [2013] is used to map to relations. Xser was the best performing system at QALD-4 and QALD-5, beating other systems by a wide margin (more than 30% absolute F-measure). According to the authors (private communication), the system achieves 69% and 39% accuracy on Free917 and WebQuestions, respectively. This is about 10% below the current state of the art on these benchmarks (see below).

4.8.3 Systems Evaluated on Free917 and WebQuestions

Sempre [Berant et al., 2013a] produces a semantic parse of a question by recursively computing logical forms corresponding to subsequences of a question. The generation is guided by identified entities in the question, a mapping of phrases to relations from the knowledge base, and a small set of composition rules. Logical forms are scored with a log-linear model and translated into a corresponding SPARQL query on Freebase. Sempre achieves 62% accuracy on Free917 and 36% accuracy on WebQuestions.

Parasempre [Berant and Liang, 2014] uses a set of fixed query patterns that are matched to each question. Each pattern is then translated back into a canonical natural language realization using a set of rules and the Freebase schema. A log-linear model chooses the realization that best paraphrases the original question. The model utilizes word vector representations and a corpus of paraphrases [Paralex, 2013]. Parasempre achieves 69% accuracy on Free917 and 40% accuracy on WebQuestions.

²³<https://github.com/dnmilne/wikipediaminer>

Graphparser [Reddy, Lapata, and Steedman, 2014] uses distant supervision to generate learning examples (questions and their answer) from natural language sentences. Intuitively, this is achieved by removing an identified entity from a sentence and reformulating the sentence as a question for that entity. To translate a question, an existing CCG parser (a kind of constituent parser) is used to retrieve a logical form. This logical form is then matched to a graph in which identified entities and relations are mapped to Freebase. Graphparser was evaluated on a subset of Freebase, where it achieves 65% accuracy on Free917 and 41% accuracy on WebQuestions.

Bordes, Chopra, and Weston [2014] take an alternative approach that involves neither named-entity recognition nor sentence parsing, and not even POS tagging. Instead, word vectors (see Section 3.4) of words and of entities and relations from Freebase are learned. This is done by using the given training data, augmented by synthetically generated question answer pairs. The idea is to learn the embeddings in such a way that the embedding of a question is close to the embedding of its answer entity. No intermediate structured query is generated. The system achieves 39% accuracy on WebQuestions.

Aqqu [Bast and Haussmann, 2015] directly constructs a SPARQL query by matching a fixed set of query patterns to the question. The patterns are matched by first identifying candidates for entity mentions in the question. Candidate queries are then generated by matching patterns on the subgraphs of these entities in the knowledge base. This way, only candidates that have an actual representation in the knowledge base are created. The candidates are ranked using a learning to rank approach. Features include the quality of entity matches and besides others, distant supervision and n -gram based approaches of matching the relations of a candidate query to the question. For entity synonyms, CrossWikis [2012] is utilized. The system achieves 76% accuracy on Free917 and 49% accuracy on WebQuestions.

STAGG [Yih et al., 2015], like Aqqu, directly constructs a SPARQL query using the knowledge base. Starting from identified entities it also incrementally constructs query candidates. To control the search space, STAGG only considers a limited number of top candidates, scored by

a learned function, for extension at each step. For scoring the candidates it also uses a learning to rank approach. In contrast to Aqqu, it uses more sophisticated techniques based on deep learning for matching relations of query candidates to the question. It also allows adding constraints to queries (e.g., *first* or *last* for dates) and, in theory, allows arbitrary patterns to be generated. In practice, however, patterns are constrained (very similar to those of Aqqu) in order to keep the search space tractable. The system achieves 53% accuracy on WebQuestions.

4.8.4 Commercial Systems

WolframAlpha can answer questions about general knowledge. As of this writing, no technical publications were available, but their FAQ²⁴ is quite informative concerning the scope and basic techniques used. On the back end side, Wolfram Alpha uses its own internal knowledge base, which is a carefully curated combination of various high-quality knowledge bases. It also uses real-time data (like weather or market prices), which is curated using heuristics. NLP techniques are used, combined with publicly available data. For example, Wikipedia is used for linguistic disambiguation (such that *the big apple* is a synonym for *NYC*). The implementation uses Mathematica as a programming language.

Facebook Graph Search²⁵ supports personalized searches on the relations between persons, places, tags, pictures, etc. An example query is *photos of my friends taken at national parks*. Results are based on the relationships between the user and her friends and their interests expressed on Facebook. Graph Search was introduced by Facebook in March 2013. It was reduced to a much restricted version (eliminating most search patterns) in December 2014, mainly due to privacy issues.

Google Search answers an increasing fraction of natural language queries from its internal knowledge base, called *Knowledge Graph*. As of this writing, the Knowledge Graph is based on Freebase (and not on the much larger *Knowledge Vault* described in Section 4.3.4) and there is no published work on how this search works.

²⁴<http://www.wolframalpha.com/faqs.html>

²⁵http://en.wikipedia.org/wiki/Facebook_Graph_Search

4.8.5 Benchmarks

Question Answering over Linked Data (QALD) [Lopez et al., 2011b; Lopez et al., 2012; Cimiano et al., 2013; Unger et al., 2014; Lopez et al., 2013; Unger et al., 2015] is an annual benchmark of manually selected natural language queries with their SPARQL equivalent. The questions are of varying complexity, for example:

Who is the mayor of Berlin?

What is the second highest mountain on Earth?

Give me all people that were born in Vienna and died in Berlin.

The name seems to imply semantic web data, but the datasets are DBpedia and MusicBrainz (see Table 2.2), which we consider as knowledge bases in this survey. For the first version of the benchmark (QALD-1) 50 training questions and 50 test questions were used. Later versions used between 50 and 100 training and test questions. Systems were evaluated by comparing the set of answers returned by a system to the answers in the ground truth (i.e., those returned by the correct SPARQL query) and computing precision and recall for each question. Averages of these on all queries and the resulting F-measure are used to compare systems globally.

The benchmark started in 2011 (QALD-1) with 2 participating groups. Since then participation has constantly increased to 7 groups for QALD-5. Later versions included questions in multiple languages and hybrid questions that require combining search on text as well as knowledge bases. The best system at QALD-4 and QALD-5 was Xser [Xu, Feng, and Zhao, 2014], described above, with an F-measure of 72% and 63%, respectively.

Free917 [Cai and Yates, 2013] is a benchmark consisting of 917 questions and their structured query (SPARQL) equivalent on Freebase. For example, *when was starry night painted* and:

```
SELECT DISTINCT ?x WHERE {
  fb:en.de_sternennacht fb:visual_art.artwork.date_completed ?x }
```

The goal is, given the question (and knowing the schema of Freebase), to automatically compute the corresponding structured query. Questions and their SPARQL equivalent were constructed manually. All

questions are such that the corresponding entities and relation indeed occur in Freebase; this makes the benchmark simpler than a real-world task with arbitrary questions from real users. 30% of the questions are explicitly marked as test questions and 70% are reserved for learning. As an evaluation metric, accuracy (the percentage of questions answered exactly as in the ground truth) is used. The current best system, Aquu [Bast and Hausmann, 2015], achieves an accuracy of 76%.

WebQuestions [Berant et al., 2013b] is a benchmark that consists of 5,810 questions and their answers from Freebase (i.e., no corresponding SPARQL query). For example, *what type of music did vivaldi write* and the answer *classical music*. In order to obtain the questions, 100,000 candidates were generated using the Google Suggest API and Amazon Mechanical Turk was used to identify those, which actually have an answer in Freebase. These questions are more realistic (i.e., more colloquial) than those of Free917, which also makes the benchmark considerably harder. 40% of the questions are used as test questions and 60% are reserved for learning. The average F-measure over all test questions is used as evaluation metric. This is computed by comparing the result set of a system to the result set in the ground truth for each question and computing individual F-measures and their average. The current best system, STAGG [Yih et al., 2015], achieves an F-measure of 53%.

4.9 Question Answering on Combined Data

Data	Combined data, as described in Section 2.1.3: text linked to a knowledge base, multiple knowledge bases, or semantic web data
Search	Natural language queries, as described in Section 2.2.3; the result is (close to) the answer one would expect from a human
Approach	A melting pot of all techniques from the previous sections; plus techniques to evaluate the confidence and combine the answers from the various sources; current approaches are still relatively simplistic, however
Strength	The ultimate “semantic search”: free-form queries on whatever data there is
Limitation	This line of research is still in its infancy; but it will be the future

In a sense, this is the ultimate “semantic search”. Users can formulate queries in natural language, and the system draws on a variety of data sources to answer it: text, knowledge bases, and combinations of the two (including semantic web data).

As of this writing, there is still little research for this scenario. In particular, we know of only one recent benchmark (the QALD-5 hybrid track with only three participants) and few notable systems; see below. This is understandable given what we have seen in the last two subsections: that natural language queries are hard already when restricting to “only” textual data or when restricting to (usually a single) knowledge base.

4.9.1 Basic Techniques

Technically, question answering on combined data is a big melting pot of techniques, in particular, those from the three previous subsections:

Question Answering from Text, *Question Answering from Knowledge Bases*, and *Keyword or Semi-Structured Search on Combined Data*, which in turn draw heavily on techniques from the previous subsections. In Section 4.9.2, we provide a longer description of the popular Watson system. Apart from Watson, there has been little research on this topic so far. In Section 4.9.3 we describe a recent system.

Commercial search engines like Google also provide question answering capabilities on both text and knowledge bases. At the time of this writing, there is no published work on how these subsystems are combined. However, answers appear to come from two different subsystems. If the answer comes from the knowledge base, the result is displayed in an infobox on the right, or as a list of entities on the top of the usual search results. If the answer comes from annotated text, it is displayed with a corresponding snippet, again on top of the usual result list. So far, there is no evidence of a deeper integration of the two kinds of data.

A survey that addresses question answering specifically for the Semantic Web is provided by Lopez et al. [2011a].

4.9.2 Watson

IBM's Watson [Ferrucci et al., 2010; Ferrucci et al., 2013] was developed to compete with human experts in the well-known Jeopardy! game show. In Jeopardy, the question is formulated as an assertion (called "claim") and the answer has to be formulated as a question. The following example clarifies that this is just an entertaining twist of classical question answering; technically the transformation of one to the other is trivial.

Classical: What drug has been shown to relieve the symptoms of ADD with relatively little side effects? Ritalin.

Jeopardy: This drug has been shown to relieve the symptoms of ADD with relatively few side effects. What is Ritalin?

The goal of Watson was to answer roughly 70% of the questions with greater than 80% precision in 3 seconds or less. This would be enough

to beat the best human experts in the game; a goal eventually reached in a much publicized show in 2011.

Watson answers questions using both text and knowledge bases. Among the text data sources are: Wikipedia, several editions of the Bible, and various encyclopedias and dictionaries. These were expanded to contain text extracted from the Web. Overall, the corpus contained 8.6 million documents with a size of 59 GB. Among the knowledge bases are: DBpedia, YAGO, and Freebase (see Table 2.2).

The Watson system consists of a pipeline of steps. Each step is very carefully designed and adjusted to the particular type and distribution of Jeopardy questions. The steps are of varying complexity and make use of state-of-the-art techniques where necessary, but also resort to simple but effective heuristics when sufficient. Here, we outline the main steps and relate them to other techniques in this survey when appropriate. For a comprehensive technical description, we refer to a special issue of the IBM Journal by Pickover [2012] consisting of a series of twelve papers (each about 10 pages) solely about Watson.

Question analysis: First, the *focus* and *lexical answer type* of the question is determined. For example, in

A new play based on this Sir Arthur Conan Doyle canine classic opened on the London stage in 2007.

the focus is *this* and the lexical answer type is *classic*. This is done using manually designed rules, e.g., “use the word *this* as focus and use its head word (*classic*) as a lexical answer type”. The rules make use of a linguistic analysis of the question, e.g., a syntactic parse, its logical structure (similar to semantic role labeling) and identified named entities; see Section 3 on *Basic NLP Tasks in Semantic Search*.

Using rules, the question is also categorized into different types, e.g., *puzzle* or *definition* question. These require slightly different approaches later on.

Relations mentioned in the question are identified as well. This is done using human-made rules as well as machine learning. The rules for about 30 relations, with 10 to 20 rules per relation, make use of identified types. For example, from *a David Lean classic* the relation

directorOf can be extracted. The corresponding rule matches if a director (*David Lean*) is used as an adjective of a *film* synonym (*classic*). The machine-learning based approach uses distant supervision (explained in Section 4.3.2) to learn relation mentions for about 7K relations from DBpedia and Wikipedia. The identified relations are simple in the sense that they only connect one entity to a relation. This is in contrast to some of the techniques in Section 4.8 on *Question Answering on Knowledge Bases*, where the goal is a formal representation of the (meaning of the) whole question.

Finally, the question is also decomposed into subquestions, which can be answered independently. For example,

This company with origins dating back to 1876 became the first U.S. company to have 1 million stockholders in 1951.

contains two major hints: that the company has “origins dating back to 1876” and that it was “the first U.S. company to have 1 million stockholders in 1951”. The decomposition is done via rules on a syntactic parse of the sentence. Answers from different subquestions are synthesized at the end with a model that is specifically trained to combine the results (lists of entities) of individual subquestions.

Hypothesis generation: After analyzing the question, the system generates candidate answers by searching multiple data sources (text and knowledge bases) independently. The focus in this step is on recall, with the assumption that later steps can weed out incorrect candidates and improve precision. A correct candidate answer not generated in this step will lead to a wrong final answer.

For search in text, standard techniques for keyword search (see Section 4.1) are applied to find documents and passages that contain keywords of the question. Candidate answers are extracted, e.g., from the title of the documents and passages using named-entity recognition. The applied techniques are similar to those of state-of-the-art systems for question answering on text (see Section 4.7).

Knowledge bases are queried for entities that are related to those mentioned in the question. These serve as additional candidate answers. If relations were identified in the question, these are also used for querying the knowledge bases. For each pair of a single entity and relation

in the question, a SPARQL query is derived and executed to retrieve additional candidate answers. In total, this phase typically generates several hundreds of candidate answers.

Soft filtering: The list of candidates obtained from the steps so far is often very long. For performance reasons, the list is filtered using lightweight machine learning, for example, based on how well the lexical answer type matches the candidate. The idea is to weed out candidates that are easy to identify as unlikely answers. Only about 100 candidate answers remain.

Hypothesis and evidence scoring: Now, evidence is collected for each remaining candidate. For example, passages that mention the answer entity along the question keywords are retrieved. Structured data is also used, e.g., in geospatial and temporal reasoning. For example, in

This picturesque Moorish city lies about 60 miles northeast of Casablanca.

the latitude and longitude of *Casablanca* can be retrieved from DBpedia and compared to candidate answers and the identified relation *northeast*.

The retrieved evidence is passed to scorers that determine the degree to which the evidence supports the candidate answer. More than 50 different scorers are used in total. They range from relatively simple string matching (between the question and the retrieved passages), to learning-based reasoning (for example, on spatial or temporal distance). According to the authors, no single algorithm dominates, but it is the ensemble that makes a difference.

Merging and ranking: In a final step, answer candidates are merged and then ranked. Merging is necessary because candidates can have different surface forms but refer to the same entity, for example, *John F. Kennedy* and *J.F.K.* This can happen because entities are retrieved from many different sources (text, DBpedia, Freebase etc.) and no canonical entity representation is enforced before this merging step.

The answer candidates are then ranked, based on the previously computed evidence scores. The question type is also taken into account.

This is important, since, for example, different features are important for factoid questions and puzzle-type questions. Ranking is done via a machine learning framework that takes as input a set of candidate answers with their evidence scores and outputs a confidence score for each candidate that indicates whether it is the final correct answer. Candidates ranked by their confidence scores are the final output of Watson.

4.9.3 Other Systems

Joshi, Sawant, and Chakrabarti [2014] answer entity-oriented (telegraphic) keyword queries on a text linked to a knowledge base (ClueWeb + FACC, see Table 2.3). Telegraphic queries are abbreviated natural language queries, for example, *first american in space*.²⁶ Given a question, the system computes a score for all possible entities, e_a , as answer. For this, the question is first split into entity, target type, relation, and contextual (everything else) segments. Then, evidence is collected from the knowledge base or text. For example, for the entity identified in the question, e_q , the relation to a possible answer entity, e_a , is retrieved from the knowledge base. A score is computed (using a language model) indicating how well the relation segment of the question expresses this knowledge-base relation. Furthermore, a text index is queried for snippets mentioning both e_q and e_a , scored by an adaptation of BM25. The final ranking incorporates the scores above as well as further evidence, like answer type information and a likelihood for the segmentation (several are possible). Overall, this is similar to the systems described in Section 4.8, but with a full-text component added to the underlying search. The system is evaluated on adapted queries from TREC (Section 4.1.3), INEX (Section 4.5.3), and WebQuestions (Section 4.8.5). On the WebQuestions dataset it achieves an nDCG@10 of 47% compared to Sempre, an approach only utilizing the knowledge base (see Section 4.8), with 45%. On the TREC and INEX questions they achieve an nDCG@10 of 54% versus 25% with Sempre.

²⁶See our discussion on the gray zone between keyword queries and natural language queries at the beginning of Section 2.2.

4.9.4 Benchmarks

The QALD series, described in Section 4.8 on *Question Answering on Knowledge Bases*, featured a *hybrid search* task in 2015. The benchmark contains ten hybrid questions, for example:

Who is the front man of the band that wrote Coffee & TV?

The correct answer requires the combination of triples with information from the textual description of the entity (both contained in DBpedia, see Table 2.2). For example, for the example question above, the information who is a front man is contained only in the text. Only five systems participated in this benchmark. The best F-measure of only 26% was achieved by the ISOFT system [Park et al., 2015].

The INEX series, described in Section 4.5.3 on *Keyword Search on Combined Data*, featured a Jeopardy! task in 2012 and 2013. However, participation was low, with only a single group in 2013.

5

Advanced Techniques used for Semantic Search

This section is about four more advanced aspects of semantic search: ranking, indexing, ontology matching and merging, and inference. They are advanced in the sense that powerful semantic search engines can be built with relatively simplistic solutions for these aspects. Indeed, this is the case for several state of the art systems and approaches from Section 4. However, when addressed properly, they can further boost result quality and/or performance.

5.1 Ranking

Many of the systems from our main Section 4 produce a list of entities as their result. In our descriptions so far, we have focused on how the set of result entities is retrieved from the respective data. In this section, we elaborate on the aspect of ranking these entities. We also (but not exclusively) include research on ranking techniques that have not been implemented as part of a full-fledged system.

The following subsections roughly correspond to the representation of the entities that should be ranked: entities associated with virtual documents (typically obtained from keyword search on combined

data; see Section 4.5), entities obtained from text linked to a knowledge base (typically obtained from keyword or semi-structured search on combined data; see Sections 4.5 and 4.6), interlinked entities (from a knowledge base or from semantic web data), and entities obtained from a structured or semi-structured search (as described in Section 4.2 and Section 4.6).

In the following, we assume basic knowledge about standard ranking techniques for document-centric keyword search on text, such as: BM25 scoring, language models, and PageRank.

5.1.1 Ranking of Entities Associated with (Virtual) Documents

A standard approach for entity search in heterogeneous data is to construct, for each entity, a virtual document consisting of (all or a selection of) text associated with the entity in the given data (typically: semantic-web data); see Section 4.5. A ranking can then be obtained by using standard techniques for keyword search in text, like BM25 or language models.

The original structure (provided by triples) does not necessarily have to be discarded. It can be preserved in a fielded index and by a ranking function like BM25F, which is an extension of BM25 by Zaragoza et al. [2004]. In comparison to standard BM25, BM25F computes a field-dependent normalized term frequency tf_f^* which, instead of document length and average document length, uses field length (l_f) and average field length ($avfl$). In addition, each field has its own “ b -parameter” B_f .

$$tf_f^* := \frac{tf_f}{1 + B_f(\frac{l_f}{avfl} - 1)}$$

The final term pseudo-frequency, that is used in the BM25 formula, is then obtained as weighted sum (field weight W_f) over the values for each field:

$$tf^* = \sum_f tf_f^* \cdot W_f$$

Originally, this improves ranking keyword queries on text by accounting for document structure (for example, with fields like *title*, *ab-*

abstract, and *body*), but this extension also applies to fields that originate from different triple predicates.

The effectiveness of BM25F for ad-hoc entity retrieval on RDF data is demonstrated by Blanco, Mika, and Vigna [2011]. Some predicates and domains are manually classified as important or unimportant (for example, *abstract* and *description* are important properties, *date* and *identifier* are unimportant). Everything not classified is treated neutrally. Important predicates have their own index field, which is then boosted in the ranking function by using a higher field weight W_f . Important domains are boosted in a separate step after the BM25F value is computed. Compared to vanilla BM25, this leads to 27% MAP instead of 18% and 48% nDCG instead of 39% on the benchmark from the SemSearch Challenge 2010 (see Section 4.5.3).

Neumayer, Balog, and Nørnvåg [2012] show that creating language models from virtual documents can outperform the fielded approach above. An entity e is ranked by the probability $p(q|e)$ of generating the query q . Different possibilities for computing this model are suggested. An *unstructured* model estimates term probabilities from the virtual document of each entity. A *structured* model groups each entity's predicates (groups are attributes, names, incoming and outgoing relations) and computes a model for each group's virtual document. The final score for an entity is a linear interpolation of these models with manually chosen weights. Experiments on the benchmarks from the SemSearch Challenges 2010 and 2011 show that the unstructured model outperforms previous state of the art but is in turn outperformed by the structured model. The authors also suggest a *hierarchical* language model that is supposed to preserve the structure (i.e. predicates) associated with entities, but the model fails to improve on previous results.

Herzig et al. [2013] also rank entities (virtual documents) using language models (LM). The work addresses two problems: identifying entities that refer to the same real-world entity, and ranking for federated search (where results from multiple sources have to be combined). The LM for an entity consists of multiple standard LMs, one for each of its attributes. A similarity distance between two entities is com-

puted by the Jensen-Shannon divergence (JSD) between the LMs for attributes that both entities have in common. If this distance is below a threshold, two entities are considered the same. Ranking for federated search works as follows. The query is issued to the multiple sources and the ranked results are used to create a new virtual document. This virtual document consists of the contents of the virtual documents of each result entity (weighted by rank). Then, a language model for the query is computed from the virtual document, which serves as a form of pseudo-relevance feedback. All entities are ranked by the similarity (again using JSD) of their language model to that of the query. In a final step, identical entities are merged as determined by the procedure above or by explicit *sameAs* links.

5.1.2 Ranking of Entities from Text Linked to a Knowledge Base

Search on text linked to a knowledge base (see Sections 4.5 and 4.6) provides two sources of signals for ranking result entities: from the matching text, and from the entity's entry in the knowledge base. However, it is not obvious how they should be combined for maximum effectiveness.

Fang et al. [2009] provide a simple but effective ranking approach for keyword queries on text, which won the TREC Entity Track in 2009 (see Section 4.5.3). In the first step, answer candidates are extracted from results (using basic NER techniques as described in Section 3) of a query to Google. This establishes the link between the text and an entity's representation in the knowledge base. In addition, a supporting passage (for an occurrence in text: the sentence; for an occurrence in a table: elements from the same column, column header, and sentence preceding the table) is extracted for each entity. Entities are then ranked by the product of three relevance probabilities: of the containing document to the query, of the containing passage, and of the entity. Document relevance is computed using a standard language model. Passage relevance is computed as a sum of similarity scores (from WordNet) between all pairs of words in the passage and the query. Entity relevance is computed as the frequency of the first query keyword (which usually corresponds to the target type, see the example above) in the entity's list of Wikipedia categories.

Kaptein and Kamps [2013] perform keyword search on Wikipedia (where documents naturally correspond to entities) and additionally make use of target categories that restrict the set of relevant entities. For example, for the query *works by charles rennnie mackintosh*, answers should be restricted to *buildings and structures*. These target categories are either given as part of the query, or can be derived automatically from the result set of an issued keyword query. Instead of simply filtering the result entities by the given or determined category, the authors suggest using language models that are supposed to deal with the hierarchical structure of categories. Two standard language models are precomputed: one for each entity (that is, for its Wikipedia page), and one for each category (that is, for the text from all entities in that category). The final score is a weighted combination of the two language models (how well they model the query) and an additional pseudo-relevance feedback computed via links between the Wikipedia pages of the top results. Weights for the combination are chosen manually.

Schuhmacher, Dietz, and Ponzetto [2015] adapt the learning-to-rank approach to keyword entity search on text, with entities already linked to a knowledge base. For a given query, the entities from the (top) documents matching the keyword query are retrieved and a feature vector is constructed for each query-entity pair. There are two groups of features: text features (for example, the occurrence count of the entity in text) and query features (for example, does the entity, or an entity closely connected in the knowledge base, occur in the query). The training set consists of queries with a given set of relevant entities. A support vector machine is used for learning to rank. It uses a linear kernel that is enhanced with a function to compute the similarity between two entities via their *relatedness* in the given knowledge base. This is supposed to provide a form of pseudo-relevance feedback. The approach is evaluated on an own benchmark that is constructed from the TREC Robust and Web benchmarks (see Section 4.1.3).

5.1.3 Ranking of Interlinked Entities

A knowledge base, or a collection of interlinked knowledge bases as in semantic-web data, can be viewed as a graph with the entities as nodes and the edges as relations; see Section 2.1.2 on *Knowledge Bases*.

Swoogle [Ding et al., 2004] adapts PageRank, the well-known algorithm to compute query-independent importance scores for web pages, to semantic-web data. Links between so-called *semantic web documents* (RDF documents defining one or more entities) are weighted differently depending on their type. Swoogle classifies links into four categories (*imports*, *uses-term*, *extends*, and *asserts*). This is done by manually defining which original RDF properties belong to which category. For each of these types, a weight is assigned manually. The PageRank transition probability from node i to node j then depends on the sum of weights of all links from i to j . This approach is feasible because only a relatively small number of different link types is considered.

ObjectRank [Balmin, Hristidis, and Papakonstantinou, 2004] adapts PageRank to keyword search on databases. The computed scores depend on the query. Intuitively, a random surfer starts at a database object that matches the keyword and then follows links pertaining to foreign keys. Edge weights are, again, based on types and assigned manually. For example, in a bibliographic database, citations are followed with high probability. Like this, the approach allows relevant objects to be found even if they do not directly mention the query keyword.

Agarwal, Chakrabarti, and Aggarwal [2006] combine PageRank with the learning to rank approach. The input is a knowledge graph (think of the semantic web) and a partial preference relation on the set of entities (think of a user more interested in some entities than in others). The goal is to learn edge weights such that the scores computed by the PageRank process (with transition probabilities proportional to these edge weights) reflect the given user preferences. Two scenarios are considered: individual weights for each edge, and one weight per edge type (predicate). For the first scenario, the problem is formulated as a constrained flow optimization problem, where the constraints come from the user preferences. For the second scenario, the problem is solved using gradient descent optimization, where the loss function

captures the user preferences (approximately only, so that it becomes differentiable).

TripleRank [Franz et al., 2009] extends the HITS algorithm to semantic-web data. HITS is a variant of PageRank, which computes hub and authority scores for each node of a sub-graph constructed from the given query. For TripleRank, the subgraph is represented as a 3D tensor where each slice is the (entity-entity) adjacency matrix for one predicate. Standard 3D tensor decomposition then yields n principal factors (corresponding to the singular values in the 2D case) and three 2D matrices with n columns each. One of these matrices can be interpreted as the underlying “topics” of the subgraph (expressed in terms of relations). The entries in the other two matrices can be interpreted as hub and authority scores, respectively, of each entity in the subgraph with respect to the identified topics.

5.1.4 Ranking of Entities Obtained from a Knowledge Base Search

SPARQL queries have precise semantics and, like SQL, the language provides an ORDER BY attribute for an explicit ranking of the result set; see Section 4.2 on *Structured Search in Knowledge Bases*. Still, there are scenarios, where a ranking according to “relevance”, as we know it from text search, is desirable.

Elbassuoni et al. [2009] construct language models for SPARQL queries with support for keyword matching in literals. The language model for the query is defined as a probability distribution over triples from the knowledge base that match triples from the query. The language model for a result graph is straightforward: it has probability 1 or 0 for each triple, depending on whether that triple is present in the result graph (with some smoothing). Results are then ranked by their Kullback-Leibler (KL) divergence to the query.

Broccoli [Bast et al., 2012] ranks result entities using a combination of popularity scores for entities and frequency scores obtained from its interactive query suggestions. For example, a simple query for *Scientist* simply ranks all scientists in the indexed knowledge base by their popularity. But the query *Scientist occurs-with information retrieval* ranks scientist according to how frequently they co-occur with the words *in-*

formation retrieval in the given text collection. Suggestions are ranked in a similar way. For example, the suggestions for predicates for *Scientist* are ranked by how many scientists have that particular predicate. This simple ranking provided average precision at 10 and MAP scores of 67-81% and 42-44%, respectively, on two benchmarks (TREC Entity Track 2009 and Wikipedia).

Bast, Buchhold, and Haussmann [2015] present an approach to compute relevance scores for triples from type-like relations. Such a score measures the degree to which an entity “belongs” to a type. For example, one would say that Quentin Tarantino is more of a film director or screenwriter than an actor. Such scores are essential in the ranking of entity queries, e.g., “american actors” or “quentin tarantino professions”. To compute the scores, each entity and type is associated with text. The text for entities is derived via linking to their occurrences in Wikipedia. Text for entire types is derived from entities that have only one entry in the particular relation. For the example above, text for the *profession actor* is derived from entities that only have the profession *actor*. Scores are then computed by comparing the text for an entity to that for each type. For this, many different models are considered: standard machine learning, a weighted sum of terms based on their tf-idf values, and a generative model. The best models achieve about 80% accuracy on a benchmark where human judges were able to achieve 90% and sensible baselines scored around 60%.

5.2 Indexing

Most of the work in this survey is concerned with the quality aspect of semantic search. This section is concerned with the efficiency aspect. Note that indirectly, efficiency is also relevant for quality: a method with good result quality with a response time of minutes or worse is impractical in many application scenarios.

Following our classification in Section 4, semantic indexing works differently depending on the particular approach: Keyword search on text (Section 4.1) is handled by an inverted index. The inverted index is a well-researched data structure and important for information retrieval

in general. A discussion of its particularities is beyond the scope of this survey. Special indexes for structured search in knowledge bases are already discussed at length in Section 4.2. Section 4.3 is concerned with structured data extraction from text. Indexing is not an issue here. Systems for keyword search on knowledge bases (Section 4.4) and question answering (Sections 4.7, 4.8, and 4.9) are concerned with finding the right queries and post-processing results. The way data is indexed is adopted from other approaches. This leaves Sections 4.5 on *Keyword Search on Combined Data* and 4.6 on *Semi-Structured Search on Combined Data* where advanced indexing techniques are required.

In this section, we distinguish three basic approaches used by the systems from that section: using an inverted index for knowledge base data, semi-structured search based on an inverted index, and integrating keyword search into a knowledge base.

5.2.1 Using an Inverted Index for Knowledge Base Data

In Section 4.2 on *Structured Search in Knowledge Bases* we discussed indexing techniques for full SPARQL support. However, semantic web applications often have different requirements: (1) the data is extremely heterogeneous, so that queries with anything but the simplest of semantics are pointless; (2) predicate and object names can be very verbose, so that keyword matching (only an optional add-on for a SPARQL engine) is a must; (3) the data volume is large, so that speed is of primary concern. In such a scenario, the inverted index provides simple solutions with high efficiency.

In the simplest realization, a *virtual document* is constructed for each entity, consisting of (all or a subset of) the words from the triples with that entity as subject; see Section 4.5.1. A standard inverted index on these virtual documents then enables keyword queries which return ranked lists of entities. A typical system in this vein is Semplore [Wang et al., 2009].

A more advanced system is described by Blanco, Mika, and Vigna [2011]. They study three variants of a fielded index, implemented using MG4J (see Section 4.1.2). The variants have different trade-offs between query time, query expressibility, and result quality. In the basic

variant, there are different fields for subject, predicate and object of a triple and positional information is used to align items from the same original triple. This allows keyword matches for object and predicate names (e.g., find triples where the predicate matches *author*) at the price of a larger query time compared to vanilla BM25 indexing. In an alternative variant, there is a field for each distinct predicate. This still allows to restrict matches to a certain predicate (e.g., *foaf:author*) but keyword matches for predicates are no longer possible. In a refined variant, predicates are grouped by importance into three classes, with one field per class. This supports only keyword queries (without any structure, like in the basic approach from the previous paragraph), but with query times similar to vanilla BM25 indexing. Result quality is vastly improved due to consideration of those fields in the ranking function: 27% MAP instead of 18% and 48% nDCG instead of 39% on the benchmark from the SemSearch Challenge 2010 (see Section 4.5.3).

SIREn [Delbru, Campinas, and Tummarello, 2012] is built on Lucene and supports keywords queries that correspond to star-shaped SPARQL queries (with one entity at the center), where predicate and relation 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. As of this writing, the software is available as open source¹.

5.2.2 Semi-Structured Search Based on an Inverted Index

This is the method of choice for semi-structured search on combined data, as described in Section 4.6. Often, an inverted index is combined with techniques or even off-the-shelf software for indexing knowledge bases, such as Virtuoso. However, the extra effort to achieve an efficient combination usually happens on the side of the inverted index.

¹<https://github.com/rdelbru/SIREn>

In the most basic realization, for each occurrence of an entity from the knowledge base in the text (for example, ... *Obama* ...), we add an artificial word to the text (for example, *entity:Barack_Obama*). The inverted list for *entity:Barack_Obama* then contains all occurrences of this entity in the text. Standard inverted list operations enable queries such as *entity:Barack_Obama audience pope* (documents mentioning him in the context of an audience with the pope).

We next discuss two simple options to enhance the query expressiveness for this approach, by not just allowing concrete entities in the query, but semantic concepts ranging from types to arbitrary SPARQL queries.

One option, that is taken by KIM [Popov et al., 2004], is to compute the result for the knowledge base parts using a standard SPARQL engine, and to add this to the keyword query as a disjunction of all the result entities. This is simple but very inefficient when the number of result entities is large. Another option, that is taken by Mimir [Tablan et al., 2015], is to add further artificial words to the index, which allow direct processing of more complex queries without resorting to the SPARQL engine. For example, if in the example above we also add the artificial word *type:politician* to the index, we could efficiently answer queries such as *type:politician audience pope* (passages mentioning a politician in the context of an audience with the pope). This works for simple queries, but does not allow complex SPARQL queries. In this case, Mimir falls back to the inefficient KIM approach.

ESTER [Bast et al., 2007] solves this dilemma by adding artificial *entity:...* and selected *type:...* words to the inverted index (just like in the example above) and resorting to *joins* for all the remaining queries. These joins require additional information in the index lists: triples from the knowledge base are inserted into canonical documents for each entity. Join operations on the *entity:...* words are needed to use this information for matches outside of this canonical document. Therefore, items in the inverted lists have to contain a word id in addition to the usual document id, position, and score. However, using standard compression techniques, the index size is comparable to that of an ordinary inverted index, despite this addition.

All the approaches described so far share the major drawback that the result is inherently a list of document passages. For example, the keyword query *type:politician audience pope* yields a list of matching passages, possibly many of them with the same entity. From a usability perspective, the more natural result would be a list of entities (ideally, with the passages as result snippets). Worse than that, if this query appears as a sub-query of a more complex query (e.g., looking for entities of a certain type who co-occur with the result entities), we need the list of entities (and not matching passages) to be able to process that query.

Broccoli [Bast and Buchhold, 2013] solves this problem using a non-trivial extension of the inverted index. The main technical idea is to augment the inverted list for each word by information about the co-occurring entities. For example, for the occurrence of the word *edible* in a document containing *the stalks of rhubarb and broccoli are edible*, the inverted list for *edible* would not only contain one item with the id of that document (plus score and positional information) but also two additional items with the ids for the entities *rhubarb* and *broccoli*. Each entity occurrence hence leads to an extra item in all inverted lists that have an entry for that document. Broccoli avoids a blow-up of the index by indexing semantic contexts instead of whole documents, which at the same time improves search quality (see Section 4.6.2). The knowledge base part is handled by lists of id pairs for each relation, sorted by either side. This is reminiscent of using PSO (predicate-subject-object) and POS permutations, like for the systems from Section 4.2 on *Structured Search on Knowledge Bases*. Together, the extended inverted lists and relation permutations allow that knowledge base facts and textual co-occurrence can be nested arbitrarily in tree-shaped queries while retaining very fast query times.

5.2.3 Integrating Keyword Search into a Knowledge Base

All major SPARQL engines feature integrations of keyword search; see Section 4.2.1. There are two basic variants, depending on the desired semantics of the integration.

To realize something like Virtuoso’s *bif:contains* predicate (see Section 4.2), it suffices to pre-compute inverted lists for words in predicate names and objects. Standard inverted list operations then lead to lists of (ids of) predicates or objects, which can be processed further by the SPARQL engine. Compared to the approach described for SIREn above, the expressiveness is much larger (no longer restricted to star queries). The price is a much larger processing time for some queries. For example, the query *author john doe* requires a full scan over all triples using the approach just described. The reason is that both, the predicate and object part can match many items and that these do not correspond to ranges but lists of ids. In the example, many ids may match the keyword *author* and many ids may match *john doe*. While these individual lists of ids are both efficiently retrieved, a subsequent step towards matching triples is problematic.

To realize an index for text linked to a knowledge base, one could add an id for each document (or short passage) to the knowledge base, and add a special relation *occurs-in* (between words or entities and the id of the document they occur in). This covers the expressiveness of Broccoli, but with a much larger processing time. For example, the query *type:politician audience pope* requires a full scan over all triples of the *occurs-in* relation. Furthermore, such a relation becomes huge with larger text corpora because it contains an entry for each word occurrence in the collection. Note that adding a relation *occurs-with* between word and entity occurrences instead, doesn’t provide the same semantics. This doesn’t allow restricting multiple occurrences to the same document or context.

5.3 Ontology Matching and Merging

Most semantic search systems work with some kind of knowledge base, in particular, all the systems from Sections 4.2, 4.4, 4.5, 4.6, 4.8, and 4.9. Most of these systems assume a *single* knowledge base with a consistent schema/ontology, as defined in Section 2.1.2. However, to cover the data relevant for a given application, often several different knowledge bases need to be considered.

For example, think of an application that requires knowledge on movies as well as on books, and that there is a separate knowledge base for each of the two domains. A problem is that these knowledge bases may contain different representations of the same real-world entity. For example, *Stephen King* is likely to be present as a *novelist* in the book knowledge base and as a *screenwriter* in the movie knowledge base. To make proper use of the data, their ontologies (their classes/concepts, properties, relations) as well as their actual population (instances) should either be linked or merged. This means, for example, identifying links between identical persons, such as:

```
<movies:Stephen_King> <owl:sameAs> <books:Stephen_Edwin_King>
```

This is known as *instance matching*. Identifying links between classes, which is referred to as *ontology matching*, is also important in that context. For example, a script is a kind of written work:

```
<movies:Filmscript> <rdfs:subClassOf> <books:Written_Work>
```

Such links can be used to merge the ontologies into a single ontology in a pre-processing step. This is known as *ontology merging*. Alternatively, systems like Virtuoso, Jena, and Sesame (see Section 4.2.2) can be configured to make use of such links during query time.

These problems have been studied (with minor differences) mainly by the semantic web community and the database community. In a relational database, tables and their columns and data types make up the schema, analogously, to the defined classes, properties, and relations in an ontology. A database row or record is the equivalent of an instance in a knowledge base. Both communities make a distinction between matching schemata and matching actual instances or database records. Approaches to these tasks are very similar in both communities, so we provide corresponding pointers for further reading. In the following, we describe the general ideas used to tackle the problems. We focus on methods, that target automatic solutions. In practice, most systems integrate user feedback on some level.

5.3.1 Ontology Matching

Matching two ontologies means to determine an alignment between them. An alignment is a set of correspondences between uniquely identified elements (e.g., classes and properties) that specifies the kind of relation they are in. For example, whether two classes are equivalent, or one subsumes the other. Shvaiko and Euzenat [2013] provide a good survey on ontology matching. The database community refers to this problem as data matching. We refer to Doan and Halevy [2005] for a good survey on database related approaches.

Approaches: Approaches to ontology matching mainly make use of matching strategies that use terminological and structural data [Shvaiko and Euzenat, 2013]. Terminological data refers to string similarities of, e.g., labels and comments in the ontology. The idea is that highly similar names can indicate equivalence. Relationships between classes (e.g., part-of, is-a) make up structural data. The intuition is that classes in similar positions in the class hierarchy are more likely to be the same. In addition to that, some approaches make use of the actual instances of a knowledge base, or try to perform logical reasoning. The output of the different matchers is combined using pre-defined or learned weights to derive a decision.

Benchmarks: In 2004, the Ontology Alignment Evaluation Initiative (OAEI) started an annual benchmark to evaluate ontology matching systems [Euzenat et al., 2011a]. Each year, a set of benchmarking datasets that include reference alignments is published.² Participation was low in the beginning but since 2007 on average 17 groups participate with a peak of 23 groups in 2013. Systems can compete against each other and compare results. On a real world ontology matching task, systems have shown to give results with above 90% F-measure [Grau et al., 2013]. Shvaiko and Euzenat [2013] note that while great progress was made in the first years, progress is slowing down. They formulate a set of eleven major challenges that need to be tackled in the near future, in particular, more efficient matching and matching utilizing background knowledge.

²<http://oei.ontologymatching.org/>

5.3.2 Instance Matching

Instance matching refers to finding instances that represent the same individual or entity. In the Semantic Web, these are linked using *owl:sameAs*. In the database community, this is referred to as record linkage, duplicate record identification or detection, and entity matching (and some more). A lot of research on this problem has been done in that community. We refer to the surveys by Köpcke and Rahm [2010] and by Elmagarmid, Ipeirotis, and Verykios [2007]. Most of the approaches for instance matching are minor adaptations from those for databases [Castano et al., 2011].

Approaches: Similar to ontology matching, to match two instances, their attribute values are compared. This involves using string similarity (e.g., edit distance and extensions, and common q -grams), phonetic similarity (similar sounding field names are similar, even if they are spelled differently) or numerical similarity (difference) depending on the data type. Then, learning based techniques represent such an instance tuple as a feature vector of similarities and use a binary classifier. If no learning data is available, manually derived weights and thresholds can be used. Extensions of these methods also consider relationships to other instances, apply unsupervised learning techniques, or apply additional rules based on domain knowledge.

Benchmarks: The benchmark by the Ontology Alignment Evaluation Initiative (OAEI) contains several instance matching tasks since 2009. Different knowledge bases are provided for which identical entities need to be identified. The used knowledge bases consist of parts of real-world knowledge bases like DBpedia [2007] or Freebase [2007]. Systems have shown to provide up to 90% F-Measure on identifying identical instances in these [Euzenat et al., 2010; Euzenat et al., 2011b].

5.3.3 Ontology Merging

Instead of using several inter-linked knowledge bases, it may be desirable to merge them into a single coherent knowledge base. Merging these involves merging their schema/ontology (concepts, relations etc.) as well as merging duplicate instances. This requires, for example, re-

solving conflicting names and attribute values. Merging can be done in an *asymmetric* fashion, where one or more ontologies are integrated into a target ontology. In contrast, *symmetric* merging places equal importance on all input ontologies.

Approaches: Most work in the research so far has focused on computing alignments between ontologies. Bruijn et al. [2006] and Shvaiko and Euzenat [2013] describe some approaches that perform merging of ontologies. These usually take computed alignments between the ontologies as input and perform semi-automatic merging. For example, PROMPT [Noy and Musen, 2000] performs merging by iteratively suggesting merge operations based on heuristics to the user, applying the user-selected operation, and computing the next possible merge operations. The fact that many systems are semi-automatic makes it extremely hard to compare their performance and currently no widely accepted benchmark exists.

The problem of merging actual instances has not received much attention by the semantic web community. It has, however, been extensively studied by the database community. Bleiholder and Naumann [2008] give a good overview and present some systems on *data fusion*. Strategies for resolving conflicts, such as different values for the same attribute, are mainly rule based. Some common strategies are, for example, asking the user what to do, using values from a preferred source, or using the newest or average value.

5.4 Inference

Inference (or reasoning) means deriving information that is not directly in the data, but can be inferred from it. For example, from the facts that *Marion Moon* is an ancestor of *Buzz Aldrin* and that *Buzz Aldrin* is an ancestor of *Janice Aldrin*, one can infer that *Marion Moon* is also an ancestor of *Janice Aldrin*.

Surprisingly, only few systems make use of inference as an integral part of their approach to semantic search. One of the few examples is the question answering system by Lymba [Moldovan, Clark, and Bowden, 2007], which uses a reasoning engine in its answering process (see

Section 4.7.3 on *The PowerAnswer System*). This is in line with the survey by Prager [2006], who observes the same for question answering. We suppose that the reason for this is that current systems already struggle solving lower level problems, such as information extraction and translating a query into a formal representation (semantic parsing). These are, however, prerequisites for utilizing inference (let alone benefiting from it). Nonetheless, inference will certainly play a more important role in the future. As a result, here we focus on technical standards and components that enable researchers to perform inference.

A lot of triple stores include an inference engine. In addition to triples, these require as input a set of inference rules, for example, that the facts *A is ancestor of B*, and *B is ancestor of C* imply that *A is ancestor of C*. First, we introduce some languages that can be used to express these rules. We then describe some triple stores and engines (also referred to as reasoners) that allow inference over triples.

5.4.1 Languages

We first describe languages that are mainly used to describe the schema of an ontology. These allow expressing constraints for the facts of a knowledge base, for example, that a child cannot be born before its parents. This also allows inference, but, broadly speaking, with a focus on taxonomic problems.

RDF Schema [RDFS, 2008] is an extension of the basic RDF vocabulary. RDFS defines a set of classes and properties expressed in RDF, that provides basic features for describing the schema of an ontology. For example, using the RDFS elements *rdfs:Class* and *rdfs:subClassOf* allows declaring a hierarchy of classes. This allows inferring missing information, such as deriving missing class memberships based on the defined class hierarchy. For example, one might derive that Buzz Aldrin is a person from the fact that he is an astronaut and the definition that astronaut is a sub-class of person.

The Web Ontology Language [OWL, 2004] is a family of languages (OWL Lite, OWL DL, OWL Full) with different levels of expressiveness to describe ontologies. OWL is the successor of DAML+OIL. Like RDFS, OWL is used to describe the schema and semantics of an ontol-

ogy, but with a much larger vocabulary and more options. For example, it allows defining class equivalences and cardinality of predicates. A prominent artifact of OWL is the *owl:sameAs* predicate, which is used to link identical instances. OWL also allows expressing transitive and inverse relationships enabling more complex inference.

The OWL 2 Web Ontology Language [OWL 2, 2012] is the successor of OWL. It extends OWL (and is backwards compatible) by addressing some shortcomings in expressiveness, syntax, and other issues [Grau et al., 2008]. Like OWL, it consists of a family of sub-languages (OWL 2 EL, OWL 2 QL, OWL 2 RL) also called profiles. These trade some of their expressive power for more efficient reasoning and inference. OWL 2 RL and QL are considered appropriate for inference with large volumes of data.

Next, we describe three prominent languages, whose single purpose is the description of inference rules. They allow expressing rules that are either hard or impossible to define in the languages above.

The Rule Markup Language [RuleML, 2001] was defined by the Rule Markup Initiative, a non-profit organization with members of academia and industry. It is XML based and (in contrast to many other languages) allows reasoning over n -ary relations. RuleML has provided input to SWRL as well as RIF (see below).

The Semantic Web Rule Language [SWRL, 2004] uses a subset of OWL DL and RuleML. It extends the syntax of OWL but is more expressive than OWL alone.

The Rule Interchange Format [RIF, 2010] was designed primarily to facilitate rule exchange. It consists of three different dialects: Core, Basic Logic Dialect (BLD), and Production Rule Dialect (PRD). RIF is an XML language and specified to be compatible with OWL and RDF. It also covers most features of SWRL.

5.4.2 Inference Engines

Most triple stores or RDF frameworks include a reasoning component. We introduce a few prominent examples and describe their features.

A widely used benchmark for evaluating OWL reasoners is the Leigh University Benchmark (LUBM) [Guo, Pan, and Heflin, 2005].

It measures, besides other things, performance and soundness of inference capabilities. The University Ontology Benchmark (UOBM) is an extension thereof, focusing on a better evaluation of inference and scalability [Ma et al., 2006]. Because results change frequently with different hardware and software versions, we don't provide them here. Current results are usually available via, e.g., <http://www.w3.org/wiki/RdfStoreBenchmarking> or the provider of the triple store.

We already introduced the triple stores of Virtuoso, Jena, and Sesame in Section 4.2 on *Structured Search in Knowledge Bases*. Virtuoso also includes an inference engine that is able to reason on a subset of the OWL standard (e.g., *owl:sameAs*, *owl:subClassOf*).

The triple store of Jena, TDB, also supports OWL and RDFS ontologies. It comes with a set of inference engines for RDFS and OWL definitions as well as custom rules. An API allows integration of third-party or custom reasoners.

Sesame includes a memory and disk-based RDF store. It provides inference engines for RDFS and custom rules. Additional reasoners can be integrated via an API.

GraphDB³, formerly OWLIM, is a product by OntoText, also available as a free version. It includes a triple store, an inference engine, and a SPARQL query engine. GraphDB can be plugged into Sesame to provide a storage and inference back end. It can reason over RDFS, (most of) SWRL, and several OWL dialects (including OWL-2 RL).

Pellet [Sirin et al., 2007] is an open source OWL 2 reasoner written in Java. It can be integrated with different Frameworks, for example, Apache Jena. Pellet supports OWL 2 as well as SWRL rules.

OpenCyc⁴ is a knowledge base platform developed by Cyc. It provides access to an ontology containing common sense knowledge and includes an inference engine. The inference engine is able to perform general logical deduction. OpenCyc uses its own rule language CycL, which is based on first-order logic.

³<http://www.ontotext.com/products/ontotext-graphdb/>

⁴<http://www.cyc.com/platform/opencyc/>

6

The Future of Semantic Search

In this final section, let us briefly review the present state of the art in semantic search, and let us then dare to take a look into the near and not so near future. Naturally, the further we look into the future, the more speculative we are. Time will tell how far we were off.

6.1 The Present

Let us quickly review the best results on the latest benchmarks from the various subsections of our main Section 4: an $nDCG@20$ of about 30% for keyword search on a large web-scale corpus (Section 4.1); an F1 score of around 40% for filling the slots of a given knowledge base from a given text corpus (Section 4.3); a MAP score of around 25% for keyword search on a large knowledge base (Section 4.4); an $nDCGD@R$ of about 30% for entity keyword search on a large web-scale corpus (Section 4.5); an $nDCG$ of about 50% for semi-structured search on an annotated Wikipedia corpus (Section 4.6); an F1 score of around 50% for natural-language list questions on text (Section 4.7); an F1 score of around 50% for natural language questions on a knowledge base (Section 4.8); and a similar score for the same kind of questions on combined data (Section 4.9).

The results for the basic NLP benchmarks are in a similar league (considering that the tasks are simpler): an F1 score of around 75% for large-scale named-entity recognition and disambiguation (Section 3.2), a weighted F1 score (called Freval) of around 55% for sentence parsing (Section 3.3), and an accuracy of about 70% for word analogy using word vectors (Section 3.4).

These are solid results on a wide array of complex tasks, based on decades of intensive research. But they are far from perfect. In particular, they are far from what humans could achieve with their level of understanding, if they had sufficient time to search or process the data.

Let us look at three state-of-the-art systems for the scenarios from the last three subsections of Section 4: PowerAnswer (Section 4.7), Wolfram Alpha (Section 4.8), and Watson (Section 4.9). It is no coincidence that these systems share the following characteristics: (1) the main components are based on standard techniques that have been known for a long time, (2) the components are very carefully engineered and combined together, and (3) the “intelligence” of the system lies in the selection of these components and their careful engineering and combination.¹ It appears that using the latest invention for a particular sub-problem does not necessarily improve the overall quality for a complex task; very careful engineering is more important. See also the critical discussion at the end of Section 4.1.

The Semantic Web, envisioned fifteen years ago, now exists, but plays a rather marginal role in semantic search so far. It is employed in some very useful basic services, like an e-commerce site telling a search robot about the basic features of its products in a structured way. But the Semantic Web is nowhere near its envisioned potential (of providing explicit semantic information for a representative portion of the Web). Results on benchmarks that use real semantic web data are among the weakest reported in our main Section 4.

Machine learning plays an important role in making the current systems better, mostly by learning the best combination of features

¹The same can probably be said about a search engine like Google. The details are not published, but telling from the search experience this is very likely so.

(many of which have been used in rule-based or manually tuned systems before) automatically. Important sources of training data are past user interactions (in particular, clickthrough data), crowdsourcing (to produce larger amounts of training data explicitly, using a large human task force), and huge unlabelled text corpora (which can be used for distant supervision). The results achieved with these approaches consistently outperform previous rule-based or manually tuned approaches by a few percent, but also not more. Also note that this is a relatively simple kind of learning, compared to what is probably needed to “break the barrier”, as discussed in Section 6.3 below.

Interestingly, modern web search engines like Google provide a much better user experience than what is suggested by the rather mediocre figures summarized above. In fact, web search has improved dramatically over the last fifteen years. We see three major reasons for this. First, the user experience in web search is mainly a matter of high precision, whereas the results above consider recall as well. Second, web search engines have steadily picked up and engineered to perfection the standard techniques over the years (including basic techniques like error correction, but also advanced techniques like learning from clickthrough data, which especially helps popular queries). Third, a rather trivial but major contributing factor is the vastly increased amount of content. The number of web pages indexed by Google has increased from 1 billion in 2000 to an estimated 50 billion in 2015 (selected from over 1 trillion URLs). For many questions that humans have, there is now a website with an answer to that question or a slight variant of it, for example: Stack Overflow (programming) or Quora (general questions about life). Social platforms like Twitter or Reddit provide enormous amounts of informative contents, too.

6.2 The Near Future

Over the next years, semantic search (along the lines of the systems we have described in Section 4) will mature further. The already large amount of text will grow steadily. The amount of data in knowledge bases will grow a lot compared to now.

Knowledge bases will be fed more and more with structured data extracted from the ever-growing amount of text. The basic techniques will be essentially those described in Section 4.3, but elaborated further, applied more intelligently, and on more and more data with faster and faster machines. This extraction will be driven by learning-based methods, based on the basic NLP methods explained in Section 3. Data from user interaction will continue to provide valuable training data. Data from the Semantic Web might provide important training information, too (either directly or via distant supervision).

The combination of information from text and from knowledge bases will become more important. The current state of the art in systems like Watson or Google Search is that the text and the knowledge base are processed in separate subsystems (often with the knowledge base being the junior partner), which are then combined post hoc in a rather simple way. The two data types, and hence also the systems using them, will grow together more and more. Large-scale annotation datasets like FACC (see Table 2.3) and the systems described in Section 4.6 on *Semi-Structured Search on Combined Data* already go in that direction. The meager contents of Section 4.9 on *Question Answering on Combined Data* show that research in this area is only just beginning. We will see much more in this direction, in research as well as in the large commercial systems.

6.3 The Not So Near Future

The development as described so far is bound to hit a barrier. That barrier is an actual *understanding* of the meaning of the information that is being sought. We said in our introduction that semantic search is search with meaning. But somewhat ironically, all the techniques that are in use today (and which we described in this survey) merely simulate an understanding of this meaning, and they simulate it rather primitively.

One might hope that with a more and more refined such “simulation”, systems based on such techniques might converge towards something that could be called real understanding. But that is not how progress has turned out in other application areas, notably: speech

recognition (given the raw audio signal, decode the words that were uttered), image classification (given the raw pixels of an image, recognize the objects in it), and game play (beat Lee Sedol, a grandmaster of Go). Past research in all these areas was characterized by approaches that more or less explicitly “simulate” human strategy, and in all these approaches eventually major progress was made by deep neural networks that learned good “strategies” themselves, using only low-level features, a large number of training examples, and an even larger number of self-generated training examples (via distant supervision on huge amounts of unlabelled data or some sort of “self play”).

Natural language processing will have to come to a similar point, where machines can compute rich semantic representations of a given text themselves, without an explicit strategy prescribed by humans. It seems that the defining characteristics of such representations are clear already now: (1) they have to be so rich as to capture all the facets of meaning in a human sense (that is, not just POS tags and entities and grammar, but also what the whole text is actually “about”); (2) they have to be hierarchical, with the lower levels of these representations being useful (and learnable) across many diverse tasks, and the higher levels building on the lower ones; (3) they are relatively easy to use, but impossible to understand in a way that a set of rules can be understood; (4) neither the representation nor the particular kind of hierarchy has to be similar to the representation and hierarchy used by human brains for the task of natural language processing.

The defining properties might be clear, but we are nowhere near building such systems yet. Natural language understanding is just so much more multifaceted than the problems above (speech recognition, image classification, game play). In particular, natural language is much more complex and requires a profound knowledge about the world on many levels (from very mundane to very abstract). A representation like word vectors (Section 3.4) seems to go in the right direction, but can at best be one component on the lowest level. The rest is basically still all missing.

Once we come near such self-learned rich semantic representations, the above-mentioned barrier will break and we will be converging to-

wards true semantic search. Eventually, we can then feed this survey into such a system and ask: *Has the goal described in the final section been achieved?* And the answer will not be: *I did not understand 'final section'.* But: *Yes, apparently ;-)*

Acknowledgements

We are very grateful to the three anonymous referees for their inspiring and thorough feedback, which has improved both the contents and the presentation of this survey considerably. And a very warm thank you to our non-anonymous editor, Doug Oard, for his infinite patience and tireless support on all levels.

Appendices

A Datasets

AQUAINT (2002). Linguistic Data Consortium, <http://catalog.ldc.upenn.edu/LDC2002T31>.

AQUAINT2 (2008). Linguistic Data Consortium, <http://catalog.ldc.upenn.edu/LDC2008T25>.

Blogs06 (2006). Introduced by [Macdonald and Ounis, 2006], http://ir.dcs.gla.ac.uk/test_collections/blog06info.html.

BTC (2009). Andreas Harth, the Billion Triples Challenge data set, <http://km.aifb.kit.edu/projects/btc-2009>.

BTC (2010). Andreas Harth, the Billion Triples Challenge data set, <http://km.aifb.kit.edu/projects/btc-2010>.

BTC (2012). Andreas Harth, the Billion Triples Challenge data set, <http://km.aifb.kit.edu/projects/btc-2012>.

ClueWeb (2009). Lemur Project, <http://lemurproject.org/clueweb09>.

ClueWeb (2012). Lemur Projekt, <http://lemurproject.org/clueweb12>.

ClueWeb09 FACC (2013). Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. FACC1: Freebase annotation of ClueWeb corpora, Version 1 (Release date 2013-06-26, Format version 1, Correction level 0), <http://lemurproject.org/clueweb09/FACC1>.

ClueWeb12 FACC (2013). Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. FACC1: Freebase annotation of ClueWeb corpora, Version 1 (Release date 2013-06-26, Format version 1, Correction level 0), <http://lemurproject.org/clueweb12/FACC1>.

- CommonCrawl (2007). Retrieved from <http://commoncrawl.org>, December 2014.
- CrossWikis (2012). Introduced by [Spitkovsky and Chang, 2012], <http://nlp.stanford.edu/data/crosswikis-data.tar.bz2>.
- DBpedia (2007). Introduced by [Lehmann, Isele, Jakob, Jentzsch, Kontokostas, Mendes, Hellmann, Morsey, Kleef, Auer, and Bizer, 2015]. Retrieved from <http://dbpedia.org>, February 2014. Statistics taken from <http://wiki.dbpedia.org/about>, January 2016.
- FAKBA1 (2015). Jeffrey Dalton, John R. Frank, Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. FAKBA1: Freebase annotation of TREC KBA Stream Corpus, Version 1 (Release date 2015-01-26, Format version 1, Correction level 0), <http://trec-kba.org/data/fakba1>.
- Freebase (2007). Introduced by [Bollacker, Evans, Paritosh, Sturge, and Taylor, 2008]. Retrieved from <http://www.freebase.com>, January 2016. Statistics taken from <http://www.freebase.com>, January 2016.
- GeoNames (2006). Retrieved from <http://www.geonames.org>, December 2014. Statistics taken from <http://www.geonames.org/ontology/documentation.html>, January 2016.
- MusicBrainz (2003). Retrieved from <http://linkedbrainz.org/rdf/dumps/20150326/>, March 2015. Statistics taken by counting items in the downloaded dataset.
- Paralex (2013). Introduced by [Fader, Zettlemoyer, and Etzioni, 2013], <http://openie.cs.washington.edu>.
- Patty (2013). Introduced by [Nakashole, Weikum, and Suchanek, 2012], <http://www.mpi-inf.mpg.de/yago-naga/patty>.
- Penn Treebank-2 (1995). Introduced by [Marcus, Santorini, and Marcinkiewicz, 1993], <https://catalog.ldc.upenn.edu/LDC95T7>.
- Penn Treebank-3 (1999). Introduced by [Marcus, Santorini, and Marcinkiewicz, 1993], <https://catalog.ldc.upenn.edu/LDC99T42>.
- Stream Corpus (2014). TREC Knowledge Base Acceleration Track, <http://trec-kba.org/kba-stream-corpus-2014.shtml>.
- UniProt (2003). Retrieved from <http://www.uniprot.org/>, July 2014. Statistics taken from <http://sparql.uniprot.org/.well-known/void>, January 2016.
- WDC (2012). Retrieved from <http://webdatacommons.org>, November 2013.

- Wikidata (2012). Retrieved from http://www.wikidata.org/wiki/Wikidata:Database_download, October 2014. Statistics taken from <https://tools.wmflabs.org/wikidata-todo/stats.php>, January 2016.
- Wikipedia LOD (2012). Introduced by [Wang, Kamps, Camps, Marx, Schuth, Theobald, Gurajada, and Mishra, 2012], <http://inex-lod.mpi-inf.mpg.de/2013>.
- YAGO (2007). Introduced by [Suchanek, Kasneci, and Weikum, 2007]. Retrieved from <http://yago-knowledge.org>, October 2009. Statistics taken from <http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/archive/>, January 2016.
- YAGO2s (2011). Introduced by [Hoffart, Suchanek, Berberich, Lewis-Kelham, Melo, and Weikum, 2011]. Retrieved from <http://yago-knowledge.org>, December 2012. Statistics taken from [Hoffart, Suchanek, Berberich, and Weikum, 2013].

B Standards

- FOAF (2000). Friend of a Friend, www.foaf-project.org.
- OWL (2004). OWL Web Ontology Language, <http://www.w3.org/TR/owl-features>.
- OWL 2 (2012). OWL 2 Web Ontology Language, <http://www.w3.org/TR/owl2-overview>.
- R2RML (2012). RDB to RDF Mapping Language, Suite of W3C Recommendations, <http://www.w3.org/TR/r2rml>.
- RDFS (2008). RDF Schema, <http://www.w3.org/TR/rdf-schema>.
- RIF (2010). Rule Interchange Format, <http://www.w3.org/TR/rif-overview>.
- RuleML (2001). Rule Markup Language, <http://ruleml.org/1.0>.
- Schema.org (2011). Google, Yahoo, Microsoft, Yandex, <http://schema.org>.
- SPARQL (2008). Query Language for RDF, W3C Recommendation, <http://www.w3.org/TR/rdf-sparql-query>.
- SQL (1986). Structured Query Language.
- SWRL (2004). A Semantic Web Rule Language, <http://www.w3.org/Submission/SWRL>.

References

- Abadi, D., P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden (2013). The design and implementation of modern column-oriented database systems. In: *Foundations and Trends in Databases* 5.3, pp. 197–280.
- Agarwal, A., S. Chakrabarti, and S. Aggarwal (2006). Learning to rank networked entities. In: *KDD*, pp. 14–23.
- Agrawal, S., S. Chaudhuri, and G. Das (2002). DBXplorer: enabling keyword search over relational databases. In: *SIGMOD*, p. 627.
- Angeli, G., S. Gupta, M. Jose, C. D. Manning, C. Ré, J. Tibshirani, J. Y. Wu, S. Wu, and C. Zhang (2014). Stanford’s 2014 slot filling systems. In: *TAC-KBP*.
- Arasu, A. and H. Garcia-Molina (2003). Extracting structured data from web pages. In: *SIGMOD*, pp. 337–348.
- Armstrong, T. G., A. Moffat, W. Webber, and J. Zobel (2009a). Has adhoc retrieval improved since 1994? In: *SIGIR*, pp. 692–693.
- Armstrong, T. G., A. Moffat, W. Webber, and J. Zobel (2009b). Improvements that don’t add up: ad-hoc retrieval results since 1998. In: *CIKM*, pp. 601–610.
- 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.
- Balakrishnan, S., A. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu (2015). Applying WebTables in practice. In: *CIDR*.

- Balmin, A., V. Hristidis, and Y. Papakonstantinou (2004). ObjectRank: authority-based keyword search in databases. In: *VLDB*, pp. 564–575.
- Balog, K. and R. Neumayer (2013). A test collection for entity search in DBpedia. In: *SIGIR*, pp. 737–740.
- Balog, K., P. Serdyukov, and A. P. de Vries (2010). Overview of the TREC 2010 Entity Track. In: *TREC*.
- Balog, K., P. Serdyukov, and A. P. de Vries (2011). Overview of the TREC 2011 Entity Track. In: *TREC*.
- Balog, K., A. P. de Vries, P. Serdyukov, P. Thomas, and T. Westerveld (2009). Overview of the TREC 2009 Entity Track. In: *TREC*.
- Balog, K., Y. Fang, M. de Rijke, P. Serdyukov, and L. Si (2012). Expertise retrieval. In: *Foundations and Trends in Information Retrieval* 6.2-3, pp. 127–256.
- Banko, M., M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni (2007). Open information extraction from the Web. In: *IJCAI*, pp. 2670–2676.
- Bast, H. and I. Weber (2006). Type less, find more: fast autocompletion search with a succinct index. In: *SIGIR*, pp. 364–371.
- Bast, H., A. Chitea, F. M. Suchanek, and I. Weber (2007). ESTER: efficient search on text, entities, and relations. In: *SIGIR*, pp. 671–678.
- Bast, H. and B. Buchhold (2013). An index for efficient semantic full-text search. In: *CIKM*, pp. 369–378.
- Bast, H., B. Buchhold, and E. Haussmann (2015). Relevance scores for triples from type-like relations. In: *SIGIR*, pp. 243–252.
- Bast, H. and E. Haussmann (2013). Open information extraction via contextual sentence decomposition. In: *ICSC*, pp. 154–159.
- Bast, H. and E. Haussmann (2014). More informative open information extraction via simple inference. In: *ECIR*, pp. 585–590.
- Bast, H. and E. Haussmann (2015). More accurate question answering on Freebase. In: *CIKM*, pp. 1431–1440.
- Bast, H., F. Bärle, B. Buchhold, and E. Haussmann (2012). Broccoli: semantic full-text search at your fingertips. In: *CoRR* abs/1207.2615.
- Bast, H., F. Bärle, B. Buchhold, and E. Haussmann (2014a). Easy access to the Freebase dataset. In: *WWW*, pp. 95–98.
- Bast, H., F. Bärle, B. Buchhold, and E. Haussmann (2014b). Semantic full-text search with Broccoli. In: *SIGIR*, pp. 1265–1266.

- Bastings, J. and K. Sima'an (2014). All fragments count in parser evaluation. In: *LREC*, pp. 78–82.
- Berant, J. and P. Liang (2014). Semantic parsing via paraphrasing. In: *ACL*, pp. 1415–1425.
- Berant, J., A. Chou, R. Frostig, and P. Liang (2013a). Semantic parsing on freebase from question-answer pairs. In: *EMNLP*, pp. 1533–1544.
- Berant, J., A. Chou, R. Frostig, and P. Liang (2013b). The WebQuestions Benchmark. In: Introduced by [Berant, Chou, Frostig, and Liang, 2013a].
- Bhalotia, G., A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan (2002). Keyword searching and browsing in databases using BANKS. In: *ICDE*, pp. 431–440.
- Bizer, C. and A. Schultz (2009). The Berlin SPARQL Benchmark. In: *IJISWIS* 5.2, pp. 1–24.
- Blanco, R., P. Mika, and S. Vigna (2011). Effective and efficient entity search in RDF data. In: *ISWC*, pp. 83–97.
- 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.
- Bleiholder, J. and F. Naumann (2008). Data fusion. In: *ACM Comput. Surv.* 41.1, 1:1–1:41.
- Boldi, P. and S. Vigna (2005). MG4J at TREC 2005. In: *TREC*.
- 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.
- Bordes, A., S. Chopra, and J. Weston (2014). Question answering with sub-graph embeddings. In: *CoRR* abs/1406.3676.
- Bordes, A. and E. Gabrilovich (2015). Constructing and mining web-scale knowledge graphs: WWW 2015 tutorial. In: *WWW*, p. 1523.
- 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.
- Bruijn, J. de, M. Ehrig, C. Feier, F. Martín-Recuerda, F. Scharffe, and M. Weiten (2006). Ontology mediation, merging and aligning. In: *Semantic Web Technologies*, pp. 95–113.
- Bruni, E., N. Tran, and M. Baroni (2014). Multimodal Distributional Semantics. In: *JAIR* 49, pp. 1–47.

- 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.
- Cai, Q. and A. Yates (2013). Large-scale semantic parsing via schema matching and lexicon extension. In: *ACL*, pp. 423–433.
- Carlson, A., J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., and T. M. Mitchell (2010). Toward an architecture for never-ending language learning. In: *AAAI*, pp. 1306–1313.
- Carmel, D., M.-W. Chang, E. Gabrilovich, B.-J. P. Hsu, and K. Wang (2014). ERD’14: entity recognition and disambiguation challenge. In: *SIGIR*, p. 1292.
- Castano, S., A. Ferrara, S. Montanelli, and G. Varese (2011). Ontology and instance matching. In: *Knowledge-Driven Multimedia Information Extraction and Ontology Evolution*, pp. 167–195.
- Charniak, E. (2000). A maximum-entropy-inspired parser. In: *ANLP*, pp. 132–139.
- Chen, D. and C. D. Manning (2014). A fast and accurate dependency parser using neural networks. In: *ACL*, pp. 740–750.
- Cheng, G., W. Ge, and Y. Qu (2008). Falcons: searching and browsing entities on the semantic web. In: *WWW*, pp. 1101–1102.
- Choi, J. D., J. R. Tetreault, and A. Stent (2015). It depends: dependency parser comparison using a web-based evaluation tool. In: *ACL*, pp. 387–396.
- Cimiano, P., V. Lopez, C. Unger, E. Cabrio, A.-C. N. Ngomo, and S. Walter (2013). Multilingual question answering over linked data (QALD-3): lab overview. In: *CLEF*, pp. 321–332.
- Coffman, J. and A. C. Weaver (2010). A framework for evaluating database keyword search strategies. In: *CIKM*, pp. 729–738.
- Coffman, J. and A. C. Weaver (2014). An empirical performance evaluation of relational keyword search techniques. In: *TKDE* 26.1, pp. 30–42.
- Cornolti, M., P. Ferragina, M. Ciaramita, H. Schütze, and S. Rüd (2014). The SMAPH system for query entity recognition and disambiguation. In: *ERD*, pp. 25–30.
- Corro, L. D. and R. Gemulla (2013). ClausIE: clause-based open information extraction. In: *WWW*, pp. 355–366.
- Craven, M., D. DiPasquo, D. Freitag, A. McCallum, T. M. Mitchell, K. Nigam, and S. Slattery (1998). Learning to extract symbolic knowledge from the world wide web. In: *AAAI*, pp. 509–516.

- Cucerzan, S. (2012). The MSR system for entity linking at TAC 2012. In: *TAC*.
- Cucerzan, S. (2007). Large-scale named entity disambiguation based on Wikipedia data. In: *EMNLP-CoNLL*, pp. 708–716.
- Cucerzan, S. (2014). Name entities made obvious: the participation in the ERD 2014 evaluation. In: *ERD*, pp. 95–100.
- Dang, H. T., D. Kelly, and J. J. Lin (2007). Overview of the TREC 2007 Question Answering Track. In: *TREC*.
- Dang, H. T., J. J. Lin, and D. Kelly (2006). Overview of the TREC 2006 Question Answering Track. In: *TREC*.
- 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.
- Dill, S., N. Eiron, D. Gibson, D. Gruhl, R. V. Guha, A. Jhingran, T. Kanungo, K. S. McCurley, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien (2003). A case for automated large-scale semantic annotation. In: *J. Web Sem.* 1.1, pp. 115–132.
- Ding, L., T. W. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs (2004). Swoogle: a search and metadata engine for the semantic web. In: *CIKM*, pp. 652–659.
- Doan, A. and A. Y. Halevy (2005). Semantic integration research in the database community: a brief survey. In: *AI Magazine*, pp. 83–94.
- Dong, X., E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmman, S. Sun, and W. Zhang (2014). Knowledge Vault: a web-scale approach to probabilistic knowledge fusion. In: *KDD*, pp. 601–610.
- 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.
- 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.
- Elmagarmid, A. K., P. G. Ipeirotis, and V. S. Verykios (2007). Duplicate record detection: a survey. In: *TKDE*, pp. 1–16.
- Etzioni, O., A. Fader, J. Christensen, S. Soderland, and Mausam (2011). Open information extraction: the second generation. In: *IJCAI*, pp. 3–10.

- Euzenat, J., A. Ferrara, C. Meilicke, J. Pane, F. Scharffe, P. Shvaiko, H. Stuckenschmidt, O. Sváb-Zamazal, V. Svátek, and C. dos Santos (2010). Results of the ontology alignment evaluation initiative 2010. In: *OM*, pp. 85–117.
- Euzenat, J., C. Meilicke, H. Stuckenschmidt, P. Shvaiko, and C. dos Santos (2011a). Ontology alignment evaluation initiative: six years of experience. In: *J. Data Semantics* 15, pp. 158–192.
- Euzenat, J., A. Ferrara, W. R. van Hage, L. Hollink, C. Meilicke, A. Nikolov, D. Ritze, F. Scharffe, P. Shvaiko, H. Stuckenschmidt, O. Sváb-Zamazal, and C. dos Santos (2011b). Results of the ontology alignment evaluation initiative 2011. In: *OM*, pp. 158–192.
- Fader, A., S. Soderland, and O. Etzioni (2011). Identifying relations for open information extraction. In: *EMNLP*, pp. 1535–1545.
- Fader, A., L. S. Zettlemoyer, and O. Etzioni (2013). Paraphrase-driven learning for open question answering. In: *ACL*, pp. 1608–1618.
- Fang, Y., L. Si, Z. Yu, Y. Xian, and Y. Xu (2009). Entity retrieval with hierarchical relevance model, exploiting the structure of tables and learning homepage classifiers. In: *TREC*.
- Ferrucci, D. A., E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefer, and C. A. Welty (2010). Building Watson: An Overview of the DeepQA Project. In: *AI Magazine* 31.3, pp. 59–79.
- Ferrucci, D. A., A. Levas, S. Bagchi, D. Gondek, and E. T. Mueller (2013). Watson: Beyond Jeopardy! In: *Artif. Intell.* 199, pp. 93–105.
- Finkel, J. R., T. Grenager, and C. D. Manning (2005). Incorporating non-local information into information extraction systems by gibbs sampling. In: *ACL*, pp. 363–370.
- Finkelstein, L., E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppín (2002). Placing search in context: the concept revisited. In: *TOIS* 20.1, pp. 116–131.
- Frank, J., S. Bauer, M. Kleiman-Weiner, D. Roberts, N. Tripuraneni, C. Zhang, C. Ré, E. Voorhees, and I. Soboroff (2013). Stream filtering for entity profile updates for TREC 2013. In: *TREC-KBA*.
- Frank, J., M. Kleiman-Weiner, D. A. Roberts, E. Voorhees, and I. Soboroff (2014). Evaluating stream filtering for entity profile updates in TREC 2012, 2013, and 2014. In: *TREC-KBA*.
- Frank, J. R., M. Kleiman-Weiner, D. A. Roberts, F. Niu, C. Zhang, C. Ré, and I. Soboroff (2012). Building an entity-centric stream filtering test collection for TREC 2012. In: *TREC-KBA*.

- Franz, T., A. Schultz, S. Sizov, and S. Staab (2009). TripleRank: ranking semantic web data by tensor decomposition. In: *ISWC*, pp. 213–228.
- Fundel, K., R. Küffner, and R. Zimmer (2007). RelEx - relation extraction using dependency parse trees. In: *Bioinformatics* 23.3, pp. 365–371.
- Gabrilovich, E. and S. Markovitch (2007). Computing semantic relatedness using Wikipedia-based Explicit Semantic Analysis. In: *IJCAI*. Vol. 7, pp. 1606–1611.
- Gaifman, H. (1965). Dependency systems and phrase-structure systems. In: *Information and Control* 8.3, pp. 304–337.
- Gövert, N., N. Fuhr, M. Lalmas, and G. Kazai (2006). Evaluating the effectiveness of content-oriented XML retrieval methods. In: *Information Retrieval* 9.6, pp. 699–722.
- Grau, B. C., I. Horrocks, B. Motik, B. Parsia, P. F. Patel-Schneider, and U. Sattler (2008). OWL 2: the next step for OWL. In: *J. Web Sem.* 6.4, pp. 309–322.
- Grau, B. C., Z. Dragisic, K. Eckert, J. Euzenat, A. Ferrara, R. Granada, V. Ivanova, E. Jiménez-Ruiz, A. O. Kempf, P. Lambrix, A. Nikolov, H. Paulheim, D. Ritze, F. Scharffe, P. Shvaiko, C. T. dos Santos, and O. Zamazal (2013). Results of the ontology alignment evaluation initiative 2013. In: *OM*, pp. 61–100.
- Guha, R. V., R. McCool, and E. Miller (2003). Semantic search. In: *WWW*, pp. 700–709.
- Guha, R., D. Brickley, and S. MacBeth (2015). Schema.org: evolution of structured data on the web. In: *ACM Queue* 13.9, p. 10.
- Guo, Y., Z. Pan, and J. Heflin (2005). LUBM: a benchmark for OWL knowledge base systems. In: *J. Web Sem.* 3, pp. 158–182.
- Halpin, H., D. Herzig, P. Mika, R. Blanco, J. Pound, H. Thompson, and D. T. Tran (2010). Evaluating ad-hoc object retrieval. In: *IWEST*.
- Hearst, M. A. (1992). Automatic acquisition of hyponyms from large text corpora. In: *COLING*, pp. 539–545.
- Herzig, D. M., P. Mika, R. Blanco, and T. Tran (2013). Federated entity search using on-the-fly consolidation. In: *ISWC*, pp. 167–183.
- Hill, F., R. Reichart, and A. Korhonen (2015). SimLex-999: Evaluating Semantic Models With (Genuine) Similarity Estimation. In: *Computational Linguistics* 41.4, pp. 665–695.

- Hoffart, J., F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum (2011). YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In: *WWW*, pp. 229–232.
- Hoffart, J., F. M. Suchanek, K. Berberich, and G. Weikum (2013). YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. In: *Artif. Intell.* 194, pp. 28–61.
- Hoffmann, R., C. Zhang, X. Ling, L. S. Zettlemoyer, and D. S. Weld (2011). Knowledge-based weak supervision for information extraction of overlapping relations. In: *ACL*, pp. 541–550.
- Hovy, E. H., M. P. Marcus, M. Palmer, L. A. Ramshaw, and R. M. Weischedel (2006). OntoNotes: the 90% solution. In: *HLT-NAACL*, pp. 57–60.
- Hristidis, V. and Y. Papakonstantinou (2002). DISCOVER: keyword search in relational databases. In: *VLDB*, pp. 670–681.
- Hua, W., Z. Wang, H. Wang, K. Zheng, and X. Zhou (2015). Short text understanding through lexical-semantic analysis. In: *ICDE*, pp. 495–506.
- Ji, H., R. Grishman, and H. T. Dang (2011). Overview of the TAC 2011 Knowledge Base Population Track. In: *TAC-KBP*.
- Ji, H., J. Nothman, and B. Hachey (2014). Overview of TAC-KBP 2014 entity discovery and linking tasks. In: *TAC-KBP*.
- Ji, H., R. Grishman, H. T. Dang, K. Griffit, and J. Ellisa (2010). Overview of the TAC 2010 Knowledge Base Population Track. In: *TAC-KBP*.
- Joachims, T. (2002). Optimizing search engines using clickthrough data. In: *KDD*, pp. 133–142.
- Joshi, M., U. Sawant, and S. Chakrabarti (2014). Knowledge graph and corpus driven segmentation and answer inference for telegraphic entity-seeking queries. In: *EMNLP*, pp. 1104–1114.
- Kaptein, R. and J. Kamps (2013). Exploiting the category structure of Wikipedia for entity ranking. In: *Artif. Intell.* 194, pp. 111–129.
- Katz, B. (1997). Annotating the world wide web using natural language. In: *RIAO*, pp. 136–159.
- Katz, B., G. C. Borchardt, and S. Felshin (2006). Natural language annotations for question answering. In: *FLAIRS*, pp. 303–306.
- Klein, D. and C. D. Manning (2002). Fast exact inference with a factored model for natural language parsing. In: *NIPS*, pp. 3–10.

- Kolomiyets, O. and M. Moens (2011). A survey on question answering technology from an information retrieval perspective. In: *Inf. Sci.* 181.24, pp. 5412–5434.
- Köpcke, H. and E. Rahm (2010). Frameworks for entity matching: a comparison. In: *DKE*, pp. 197–210.
- Le, Q. V. and T. Mikolov (2014). Distributed representations of sentences and documents. In: *ICML*, pp. 1188–1196.
- Lee, D. D. and H. S. Seung (2000). Algorithms for non-negative matrix factorization. In: *NIPS*, pp. 556–562.
- Lehmann, J., R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer (2015). DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. In: *Semantic Web* 6.2, pp. 167–195.
- Lei, Y., V. S. Uren, and E. Motta (2006). SemSearch: a search engine for the semantic web. In: *EKAW*, pp. 238–245.
- Levy, O. and Y. Goldberg (2014). Neural word embedding as implicit matrix factorization. In: *NIPS*, pp. 2177–2185.
- Levy, O., Y. Goldberg, and I. Dagan (2015). Improving Distributional Similarity with Lessons Learned from Word Embeddings. In: *TACL* 3, pp. 211–225.
- Li, G., S. Ji, C. Li, and J. Feng (2009). Efficient type-ahead search on relational data: a TASTIER approach. In: *SIGMOD*, pp. 695–706.
- Li, H. and J. Xu (2014). Semantic matching in search. In: *Foundations and Trends in Information Retrieval* 7.5, pp. 343–469.
- Limaye, G., S. Sarawagi, and S. Chakrabarti (2010). Annotating and Searching Web Tables Using Entities, Types and Relationships. In: *PVLDB* 3.1, pp. 1338–1347.
- Liu, T. (2009). Learning to rank for information retrieval. In: *Foundations and Trends in Information Retrieval* 3.3, pp. 225–331.
- Lopez, V., V. S. Uren, M. Sabou, and E. Motta (2011a). Is question answering fit for the Semantic Web?: A survey. In: *Semantic Web* 2.2, pp. 125–155.
- Lopez, V., C. Unger, P. Cimiano, and E. Motta (2011b). Proceedings of the 1st workshop on question answering over linked data (QALD-1). In: *ESWC*.
- Lopez, V., C. Unger, P. Cimiano, and E. Motta (2012). Interacting with linked data. In: *ESWC-ILD*.

- Lopez, V., C. Unger, P. Cimiano, and E. Motta (2013). Evaluating question answering over linked data. In: *J. Web Sem.* 21, pp. 3–13.
- Lund, K. and C. Burgess (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. In: *Behavior research methods, instruments, & computers* 28.2, pp. 203–208.
- Luong, T., R. Socher, and C. D. Manning (2013). Better word representations with recursive neural networks for morphology. In: *CoNLL*, pp. 104–113.
- Ma, L., Y. Yang, Z. Qiu, G. T. Xie, Y. Pan, and S. Liu (2006). Towards a complete OWL ontology benchmark. In: *ESWC*, pp. 125–139.
- Macdonald, C. and I. Ounis (2006). The TREC Blogs06 collection: Creating and analysing a blog test collection. In: *Department of Computer Science, University of Glasgow Tech Report TR-2006-224* 1, pp. 3–1.
- Manning, C. D. (2011). Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In: *CICLING*, pp. 171–189.
- Manning, C. D., M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky (2014). The Stanford CoreNLP natural language processing toolkit. In: *ACL*, pp. 55–60.
- Marcus, M. P., B. Santorini, and M. A. Marcinkiewicz (1993). Building a large annotated corpus of English: the Penn Treebank. In: *Computational Linguistics* 19.2, pp. 313–330.
- Mass, Y. and Y. Sagiv (2012). Language models for keyword search over data graphs. In: *WSDM*, pp. 363–372.
- Mausam, M. Schmitz, S. Soderland, R. Bart, and O. Etzioni (2012). Open language learning for information extraction. In: *EMNLP-CoNLL*, pp. 523–534.
- Mayfield, J., J. Artiles, and H. T. Dang (2012). Overview of the TAC 2012 Knowledge Base Population Track. In: *TAC-KBP*.
- Mayfield, J. and R. Grishman (2015). TAC 2015 Cold Start KBP Track. In: *TAC-KBP*.
- McClosky, D., E. Charniak, and M. Johnson (2006). Effective self-training for parsing. In: *HLT-NAACL*.
- Meusel, R., P. Petrovski, and C. Bizer (2014). The WebDataCommons Microdata, RDFa and Microformat dataset series. In: *ISWC*, pp. 277–292.
- Mikolov, T., W. Yih, and G. Zweig (2013). Linguistic regularities in continuous space word representations. In: *NAACL*, pp. 746–751.

- Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado, and J. Dean (2013a). Distributed representations of words and phrases and their compositionality. In: *NIPS*, pp. 3111–3119.
- Mikolov, T., K. Chen, G. Corrado, and J. Dean (2013b). Efficient estimation of word representations in vector space. In: *CoRR* abs/1301.3781.
- Miller, G. A. (1992). WordNet: A Lexical Database for English. In: *Commun. ACM* 38, pp. 39–41.
- Mintz, M., S. Bills, R. Snow, and D. Jurafsky (2009). Distant supervision for relation extraction without labeled data. In: *ACL/IJCNLP*, pp. 1003–1011.
- Mitchell, T. M., W. W. Cohen, E. R. H. Jr., P. P. Talukdar, J. Betteridge, A. Carlson, B. D. Mishra, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. A. Platanios, A. Ritter, M. Samadi, B. Settles, R. C. Wang, D. T. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling (2015). Never-ending learning. In: *AAAI*, pp. 2302–2310.
- Mitkov, R. (2014). *Anaphora resolution*. Routledge.
- Moldovan, D. I., C. Clark, and M. Bowden (2007). Lymba’s PowerAnswer 4 in TREC 2007. In: *TREC*.
- Monahan, S., D. Carpenter, M. Gorelkin, K. Crosby, and M. Brunson (2014). Populating a knowledge base with entities and events. In: *TAC*.
- Morsey, M., J. Lehmann, S. Auer, and A.-C. N. Ngomo (2011). DBpedia SPARQL benchmark - performance assessment with real queries on real data. In: *ISWC*, pp. 454–469.
- Nakashole, N., G. Weikum, and F. M. Suchanek (2012). PATTY: A taxonomy of relational patterns with semantic types. In: *EMNLP*, pp. 1135–1145.
- Neumann, T. and G. Weikum (2009). Scalable join processing on very large RDF graphs. In: *SIGMOD*, pp. 627–640.
- Neumann, T. and G. Weikum (2010). The RDF-3X engine for scalable management of RDF data. In: *VLDB J.* 19.1, pp. 91–113.
- Neumayer, R., K. Balog, and K. Nørsvåg (2012). On the modeling of entities for ad-hoc entity search in the web of data. In: *ECIR*, pp. 133–145.
- Ng, V. (2010). Supervised noun phrase coreference research: the first fifteen years. In: *ACL*, pp. 1396–1411.
- Nivre, J., J. Hall, S. Kübler, R. T. McDonald, J. Nilsson, S. Riedel, and D. Yuret (2007). The CoNLL 2007 Shared Task on dependency parsing. In: *EMNLP-CoNLL*, pp. 915–932.

- Noy, N. F. and M. A. Musen (2000). PROMPT: algorithm and tool for automated ontology merging and alignment. In: *AAAI*, pp. 450–455.
- Oren, E., R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tumarello (2008). Sindice.com: a document-oriented lookup index for open linked data. In: *IJMSO* 3.1, pp. 37–52.
- Orr, D., A. Subramanya, E. Gabrilovich, and M. Ringgaard (2013). 11 billion clues in 800 million documents: a web research corpus annotated with freebase concepts. In: *Google Research Blog*.
- Park, S., S. Kwon, B. Kim, and G. G. Lee (2015). ISOFT at QALD-5: hybrid question answering system over linked data and text data. In: *CLEF*.
- Pennington, J., R. Socher, and C. D. Manning (2014). Glove: global vectors for word representation. In: *EMNLP*, pp. 1532–1543.
- Petrov, S. and R. McDonald (2012). Overview of the 2012 shared task on parsing the web. In: *SANCL*. Vol. 59.
- Pickover, C. A., ed. (2012). *This is Watson* 56.3–4: *IBM Journal of Research and Development*.
- 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.
- Pound, J., P. Mika, and H. Zaragoza (2010). Ad-hoc object retrieval in the web of data. In: *WWW*, pp. 771–780.
- Pound, J., A. K. Hudek, I. F. Ilyas, and G. E. Weddell (2012). Interpreting keyword queries over web knowledge bases. In: *CIKM*, pp. 305–314.
- Prager, J. M. (2006). Open-domain question-answering. In: *Foundations and Trends in Information Retrieval* 1.2, pp. 91–231.
- Qi, Y., Y. Xu, D. Zhang, and W. Xu (2014). BUPT_PRIS at TREC 2014 knowledge base acceleration track. In: *TREC*.
- Radinsky, K., E. Agichtein, E. Gabrilovich, and S. Markovitch (2011). A word at a time: computing word relatedness using temporal semantic analysis. In: *WWW*, pp. 337–346.
- Reddy, S., M. Lapata, and M. Steedman (2014). Large-scale Semantic Parsing without Question-Answer Pairs. In: *TACL* 2, pp. 377–392.
- Riedel, S., L. Yao, and A. McCallum (2010). Modeling relations and their mentions without labeled text. In: *ECML PKDD*, pp. 148–163.
- Sarawagi, S. (2008). Information Extraction. In: *Foundations and Trends in Databases* 1.3, pp. 261–377.

- Schmidt, M., M. Meier, and G. Lausen (2010). Foundations of SPARQL query optimization. In: *ICDT*, pp. 4–33.
- Schuhmacher, M., L. Dietz, and S. P. Ponzetto (2015). Ranking entities for web queries through text and knowledge. In: *CIKM*, pp. 1461–1470.
- Shvaiko, P. and J. Euzenat (2013). Ontology matching: state of the art and future challenges. In: *TKDE* 25.1, pp. 158–176.
- Silvestri, F. (2010). Mining query logs: turning search usage data into knowledge. In: *Foundations and Trends in Information Retrieval* 4.1-2, pp. 1–174.
- Sirin, E., B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz (2007). Pellet: A practical OWL-DL reasoner. In: *J. Web Sem.* 5.2, pp. 51–53.
- Socher, R., J. Bauer, C. D. Manning, and A. Y. Ng (2013). Parsing with compositional vector grammars. In: *ACL (1)*, pp. 455–465.
- Spitkovsky, V. I. and A. X. Chang (2012). A cross-lingual dictionary for english wikipedia concepts. In: *LREC*, pp. 3168–3175.
- Suchanek, F. M., G. Kasneci, and G. Weikum (2007). YAGO: a core of semantic knowledge. In: *WWW*, pp. 697–706.
- Surdeanu, M. (2013). Overview of the TAC 2013 Knowledge Base Population evaluation: english slot filling and temporal slot filling. In: *TAC-KBP*.
- Surdeanu, M. and H. Ji (2014). Overview of the english slot filling track at the TAC 2014 Knowledge Base Population evaluation. In: *TAC-KBP*.
- Surdeanu, M., J. Tibshirani, R. Nallapati, and C. D. Manning (2012). Multi-instance multi-label learning for relation extraction. In: *EMNLP-CoNLL*, pp. 455–465.
- 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.
- Tonon, A., G. Demartini, and P. Cudré-Mauroux (2012). Combining inverted indices and structured search for ad-hoc object retrieval. In: *SIGIR*, pp. 125–134.
- Toutanova, K., D. Klein, C. D. Manning, and Y. Singer (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In: *HLT-NAACL*, pp. 173–180.
- Tran, T., H. Wang, and P. Haase (2009). Hermes: data web search on a pay-as-you-go integration infrastructure. In: *J. Web Sem.* 7.3, pp. 189–203.

- Tran, T., P. Cimiano, S. Rudolph, and R. Studer (2007). Ontology-based interpretation of keywords for semantic search. In: *ISWC/ASWC*, pp. 523–536.
- Trotman, A., C. L. A. Clarke, I. Ounis, S. Culpepper, M. Cartright, and S. Geva (2012). Open source information retrieval: a report on the SIGIR 2012 workshop. In: *SIGIR Forum* 46.2, pp. 95–101.
- Unbehauen, J., C. Stadler, and S. Auer (2013). Optimizing SPARQL-to-SQL rewriting. In: *IIWAS*, p. 324.
- Unger, C., L. Bühmann, J. Lehmann, A. N. Ngomo, D. Gerber, and P. Cimiano (2012). Template-based question answering over RDF data. In: *WWW*, pp. 639–648.
- Unger, C., C. Forascu, V. Lopez, A. N. Ngomo, E. Cabrio, P. Cimiano, and S. Walter (2014). Question answering over linked data (QALD-4). In: *CLEF*, pp. 1172–1180.
- Unger, C., C. Forascu, V. Lopez, A. N. Ngomo, E. Cabrio, P. Cimiano, and S. Walter (2015). Question answering over linked data (QALD-5). In: *CLEF*.
- Voorhees, E. M. (1999). The TREC-8 Question Answering Track Report. In: *TREC*.
- Voorhees, E. M. (2000). Overview of the TREC-9 Question Answering Track. In: *TREC*.
- Voorhees, E. M. (2001). Overview of the TREC 2001 Question Answering Track. In: *TREC*.
- Voorhees, E. M. (2002). Overview of the TREC 2002 Question Answering Track. In: *TREC*.
- Voorhees, E. M. (2003). Overview of the TREC 2003 Question Answering Track. In: *TREC*.
- Voorhees, E. M. (2004). Overview of the TREC 2004 Question Answering Track. In: *TREC*.
- Voorhees, E. M. and H. T. Dang (2005). Overview of the TREC 2005 Question Answering Track. In: *TREC*.
- Voorhees, E. M. and D. K. Harman (2005). *TREC: Experiment and evaluation in information retrieval*. Vol. 63. MIT press Cambridge.
- Wang, H., Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan (2009). Semplore: A scalable IR approach to search the Web of Data. In: *J. Web Sem.* 7.3, pp. 177–188.

- Wang, Q., J. Kamps, G. R. Camps, M. Marx, A. Schuth, M. Theobald, S. Gurajada, and A. Mishra (2012). Overview of the INEX 2012 Linked Data Track. In: *CLEF*.
- Wu, S., C. Zhang, F. Wang, and C. Ré (2015). Incremental Knowledge Base Construction Using DeepDive. In: *PVLDB* 8.11, pp. 1310–1321.
- Xu, K., Y. Feng, and D. Zhao (2014). Answering natural language questions via phrasal semantic parsing. In: *CLEF*, pp. 1260–1274.
- Yahya, M., K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum (2012). Natural language questions for the web of data. In: *EMNLP-CoNLL 2012*, pp. 379–390.
- Yates, A., M. Banko, M. Broadhead, M. J. Cafarella, O. Etzioni, and S. Soderland (2007). TextRunner: open information extraction on the web. In: *HLT-NAACL*, pp. 25–26.
- Yih, W., M. Chang, X. He, and J. Gao (2015). Semantic parsing via staged query graph generation: question answering with knowledge base. In: *ACL*, pp. 1321–1331.
- Yu, J. X., L. Qin, and L. Chang (2010). Keyword search in relational databases: a survey. In: *IEEE Data Eng. Bull.* 33.1, pp. 67–78.
- 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*.
- Zelenko, D., C. Aone, and A. Richardella (2003). Kernel methods for relation extraction. In: *Journal of Machine Learning Research* 3, pp. 1083–1106.
- Zenz, G., X. Zhou, E. Minack, W. Siberski, and W. Nejdl (2009). From keywords to semantic queries - Incremental query construction on the semantic web. In: *J. Web Sem.* 7.3, pp. 166–176.
- Zhang, C. (2015). DeepDive: A Data Management System for Automatic Knowledge Base Construction. PhD thesis. University of Wisconsin-Madison.
- Zhiltsov, N., A. Kotov, and F. Nikolaev (2015). Fielded sequential dependence model for ad-hoc entity retrieval in the web of data. In: *SIGIR*, pp. 253–262.
- Zhou, G., J. Su, J. Zhang, and M. Zhang (2005). Exploring various knowledge in relation extraction. In: *ACL*, pp. 427–434.
- Zhou, Q., C. Wang, M. Xiong, H. Wang, and Y. Yu (2007). SPARK: adapting keyword query to semantic search. In: *ISWC/ASWC*, pp. 694–707.

WSDM Cup 2017: Vandalism Detection and Triple Scoring*

Stefan Heindorf
Paderborn University
heindorf@uni-paderborn.de

Martin Potthast
Bauhaus-Universität Weimar
martin.potthast@uni-weimar.de

Hannah Bast
University of Freiburg
bast@informatik.uni-
freiburg.de

Björn Buchhold
University of Freiburg
buchholb@informatik.uni-
freiburg.de

Elmar Haussmann
University of Freiburg
haussmann@informatik.uni-
freiburg.de

ABSTRACT

The WSDM Cup 2017 was a data mining challenge held in conjunction with the 10th International Conference on Web Search and Data Mining (WSDM). It addressed key challenges of knowledge bases today: quality assurance and entity search. For quality assurance, we tackle the task of vandalism detection, based on a dataset of more than 82 million user-contributed revisions of the Wikidata knowledge base, all of which annotated with regard to whether or not they are vandalism. For entity search, we tackle the task of triple scoring, using a dataset that comprises relevance scores for triples from type-like relations including occupation and country of citizenship, based on about 10,000 human relevance judgments. For reproducibility sake, participants were asked to submit their software on TIRA, a cloud-based evaluation platform, and they were incentivized to share their approaches open source.

Keywords: Knowledge Base; Vandalism; Data Quality; Search

1. TASK ON VANDALISM DETECTION

Knowledge is increasingly gathered by the crowd. Perhaps the most prominent example is Wikidata, the knowledge base of the Wikimedia Foundation that can be edited by anyone, and that stores structured data similar to RDF triples. Most volunteers' contributions are of high quality, whereas some vandalize and damage the knowledge base. The latter's impact can be severe: integrating Wikidata into information systems such as search engines or question-answering systems bears the risk of spreading false information to all their users. Moreover, manually reviewing millions of contributions every month imposes a high workload on the community. Hence, the goal of this task is to develop an effective vandalism detection model for Wikidata:

Given a Wikidata revision, the task is to compute a quality score denoting the likelihood of this revision being vandalism (or similarly damaging).

*We thank Adobe Systems Inc. for sponsoring the event, and Wikimedia Germany for supporting it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

WSDM 2017 February 06-10, 2017, Cambridge, United Kingdom

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4675-7/17/02.

DOI: <http://dx.doi.org/10.1145/3018661.3022762>

Table 1: The vandalism detection evaluation datasets in terms of time period covered, revisions, sessions, items, and users as per Heindorf et al. [7]. Numbers are given in thousands.

Dataset	From	To	Revisions	Sessions	Items	Users
Training	Oct 1, 2012	Feb 29, 2016	65,010	36,552	12,401	471
Validation	Mar 1, 2016	Apr 30, 2016	7,225	3,827	3,116	43
Test	May 1, 2016	Jun 30, 2016	10,445	3,122	2,661	41

Revisions were to be scored in near real time as soon as a revision arrives, allowing for immediate action upon potential vandalism. Moreover, a model should hint at vandalism across a wide range of precision/recall points to enable use cases such as fully automatic reversion of damaging edits at high precision, as well as pre-filtering revisions at high recall and ranking them with respect to importance of being reviewed.

For the challenge, we constructed the Wikidata Vandalism Corpus 2016 (WDVC-2016),¹ an up-to-date version of the Wikidata Vandalism Corpus 2015 (WDVC-2015) [6]: it consists of user-contributed edits, excluding edits by bots, alongside annotations whether or not an edit has been reverted via the administrative roll-back feature, which is employed at Wikidata to revert vandalism and similarly damaging contributions. This way, we obtained a large-scale corpus ranging from October 2012 to June 2016, containing over 82 million revisions, 198,147 of which are labeled as vandalism. The corpus also supplies meta information that is not readily available from Wikidata, such as geolocation data of all anonymous edits as well as Wikidata revision tags originating from both the Wikidata Abuse Filter and semi-automatic editing tools. Table 1 gives an overview of the corpus. Participants were provided training data and validation data while the test data was held back until the final evaluation. To prevent teams from using information that emerged after a revision was made, we sorted all revisions by time and employed the evaluation-as-a-service platform TIRA [4]² in combination with a newly developed data server that only provides new revisions after a participant's software has reported scores for previous revisions. The setup, datasets, rules, and measures, are described in detail on <http://www.wsdm-cup-2017.org/vandalism-detection.html>.

As our main evaluation metric, we employ the area under curve of the receiver operating characteristic because it is the de facto standard for imbalanced learning tasks and enables a comparison to state-of-the-art vandalism detectors [7]. For informational purposes, we compute the area under the precision-recall curve, too.

The final evaluation results will be published in the workshop proceedings of the WSDM Cup 2017 [5].

¹Available from <http://www.wsdm-cup-2017.org/vandalism-detection.html>

²<http://www.tira.io>

2. TASK ON TRIPLE SCORING

Knowledge bases allow queries that express the search intent precisely. For example, we can easily formulate a query that gives us precisely a list of all *American actors* in a knowledge base. Note the fundamental difference to full-text search, where keyword queries are only approximations of the actual search intent, and thus result lists are typically a mix of relevant and irrelevant hits.

But even for result sets containing only relevant items, a ranking of the contained items is often desirable. One reason is similar as in full-text search: when the result set is very large, we cannot look at all items and thus want the most “interesting” items first. But even for small result sets, it is useful to show the inherent order of the items in case there is one. We give two examples. The numbers refer to a sanitized dump of Freebase from June 29, 2014; see [1].

Example 1 (American actors): Consider the query that returns all entities that have *Actor* as their profession and *American* as their nationality. On the latest version of the Freebase dataset, this query has 64,757 matches. A straightforward ranking would be by popularity, as measured, e.g., by counting the number of occurrences of each entity in a reference text corpus. Doing that, the top-5 results for our query look as follows (the first result is G. W. Bush):

George Bush, Hillary Clinton, Tim Burton, Lady Gaga, Johnny Depp

All five of these are indeed listed as actors in Freebase. This is correct in the sense that each of them appeared in a number of movies, and be it only in documentary movies as themselves or in short cameo roles. However, Bush and Clinton are known as politicians, Burton is known as a film director, and Lady Gaga as a musician. Only Johnny Depp, number five in the list above, is primarily an actor. He should be ranked before the other four.

Example 2 (professions of a single person): Consider all professions by Arnold Schwarzenegger. Freebase lists 10 entries:

Actor, Athlete, Bodybuilder, Businessperson, Entrepreneur, Film Producer, Investor, Politician, Television Director, Writer

Again, all of them are correct in a sense. For this query, ranking by “popularity” (of the professions) makes even less sense than for the query from Example 1. Rather, we would like to have the “main” professions of that particular person at the top. For Arnold Schwarzenegger that would be: *Actor, Politician, Bodybuilder*. Note how we have an ill-defined task here: it is debatable whether Arnold Schwarzenegger is more of an actor or more of a politician. But he is certainly more of an actor than a writer.

2.1 Task Definition

The task is to compute relevance scores for triples from type-like relations. The following definition is adapted from [2]:

Given a list of triples from two type-like relations (profession and nationality), for each triple compute an integer score from 0..7 that measures the degree to which the subject belongs to the respective type (expressed by the predicate and object).

Here are four example scores, related to the example queries above:

<i>Tim Burton</i>	<i>profession</i>	<i>Actor</i>	2
<i>Tim Burton</i>	<i>profession</i>	<i>Director</i>	7
<i>Johnny Depp</i>	<i>profession</i>	<i>Actor</i>	7
<i>A. Schwarzenegger</i>	<i>profession</i>	<i>Actor</i>	6

An alternative, more intuitive way of expressing this notion of “degree” is: how “surprised” would we be to see *Actor* in a list of professions of, say, Arnold Schwarzenegger (a few people would be, most would not). This formulation is also used in the crowdsourcing task which we designed to acquire human judgments for the ground truth used in our evaluation.

2.2 Datasets

Participants were provided a knowledge base in the form of 818,023 triples from two Freebase relations: *profession* and *nationality*. Overall, these triples contained 385,426 different subjects, 200 different professions, and 100 different nationalities.

We constructed a ground truth for 1,387 of these triples (1,028 profession, 359 nationality). For each triple we obtained 7 binary relevance judgments from a carefully implemented and controlled crowdsourcing task, as described in [2]. This gives a total of 9,709 relevance judgments. For each triple, the sum of the binary relevance judgments yields the score.

About half of this ground truth (677 triples) was made available to the participants as training data. This was useful for understanding the task and the notion of “degree” in the definition above. However, the learning task was still inherently unsupervised, because the training data covers only a subset of all professions and nationalities. Participants were allowed to use arbitrary external data for unsupervised learning. For convenience, we provided 33,159,353 sentences from Wikipedia with annotations of the 385,426 subjects. For each subject from the ground truth, there were at least three sentences (and usually many more) with that subject annotated.

The setup, datasets, rules, and measures, are described in detail on <http://www.wsdm-cup-2017.org/triple-scoring.html>.

2.3 Performance Measures

Three quality measures were applied to measure the quality of participating systems with respect to our ground truth:

Accuracy: the percentage of triples for which the score (an integer from the range 0..7) differs by at most 2 (in either direction) from the score in the ground truth.

Average score difference: the average (over all triples in the ground truth) of the absolute difference of the score computed by the participating system and the score from the ground truth.

Kendall’s Tau: a ranked-based measure which compares the ranking of all the professions (or nationalities) of a person with the ranking computed from the ground truth scores. The handling of items with equal score is described in [2, Section 5.1] and under the link above.

Note that the Accuracy measure can only increase (and never decrease) when all scores 0 and 1 are rounded up to 2, and all scores 6 and 7 are rounded down to 5. For reasons of fairness, we therefore applied this simple transformation to all submissions when comparing with respect to Accuracy.

The final evaluation results will be published in the workshop proceedings of the WSDM Cup 2017 [3].

References

- [1] H. Bast, F. Bärle, B. Buchhold, and E. Hauffmann. Easy access to the freebase dataset. In *WWW*, pages 95–98, 2014.
- [2] H. Bast, B. Buchhold, and E. Haussmann. Relevance scores for triples from type-like relations. In *SIGIR*, pages 243–252, 2015.
- [3] H. Bast, B. Buchhold, and E. Haussmann. Overview of the Triple Scoring Task at WSDM Cup 2017. *To appear*, 2017.
- [4] T. Gollub, B. Stein, and S. Burrows. Ousting Ivory Tower Research: Towards a Web Framework for Providing Experiments as a Service. In *SIGIR*, pages 1125–1126, 2012.
- [5] S. Heindorf, M. Potthast, G. Engels, and B. Stein. Overview of the Wikidata Vandalism Detection Task at WSDM Cup 2017. *To appear*, 2017.
- [6] S. Heindorf, M. Potthast, B. Stein, and G. Engels. Towards Vandalism Detection in Knowledge Bases: Corpus Construction and Analysis. In *SIGIR*, pages 831–834, 2015.
- [7] S. Heindorf, M. Potthast, B. Stein, and G. Engels. Vandalism Detection in Wikidata. In *CIKM*, pages 327–336, 2016.

QLever: A Query Engine for Efficient SPARQL+Text Search

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

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

ABSTRACT

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

CCS CONCEPTS

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

KEYWORDS

SPARQL+Text; Efficiency; Indexing

1 INTRODUCTION

This paper is about efficient search in a knowledge base combined with text. For the purpose of this paper, a knowledge base is a collection of subject-predicate-object triples, where consistent identifiers are used for the same entities. For example, here are three triples from Freebase¹, the world's largest open general-purpose knowledge base, which we also use in our experiments:

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

A knowledge base enables queries that express the search intent precisely. For example, using SPARQL (the de facto standard query

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM'17, November 6–10, 2017, Singapore.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4918-5/17/11...\$15.00
DOI: <https://doi.org/10.1145/3132847.3132921>

language for knowledge bases), we can easily search for all astronauts and their nationalities as follows:

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

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

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

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

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

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

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

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

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

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

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

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

1.1 Contributions

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

- A query engine, called QLever, for a knowledge base linked to a text corpus as described above, which efficiently supports the predicates *ql:contains-word* and *ql:contains-entity* and which is also efficient (and competitive) for pure SPARQL queries.

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

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

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

- A new benchmark of 159 SPARQL+Text queries from 12 categories, including pure SPARQL queries and real-world queries adapted from the SemSearch Challenge [8].

- A performance evaluation on this benchmark, with a comparison against three state-of-the-art query engines. QLever’s query times are fastest for the pure SPARQL queries and fastest by several orders of magnitude for the SPARQL+Text queries. On the large Freebase+ClueWeb dataset, QLever achieves subsecond query times even for complex SPARQL+Text queries; see Section 5.

- Efficient support for convenient text-search features, including: text snippets as part of the result (see the *TEXT(?t)* above), scores for ranking (see the *SCORE(?t)* above), and a *TEXTLIMIT* operator for limiting the number of text snippets.

- Technical contributions are: index layouts for efficient scan and text operations; query planning for SPARQL+Text queries based on dynamic programming; novel heuristics for result size estimation.

- All code, benchmark queries and our dataset are open source and available on GitHub (<https://github.com/Buchhold/QLever>). QLever can easily be set up for any knowledge base linked to a text corpus as described above.

2 RELATED WORK

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

2.1 Systems for a KB linked to a text corpus

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

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

2.2 SPARQL engines

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

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

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

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

SPARQL queries can be rewritten to SQL [12] and all the big RDBMSs now also provide support for SPARQL. A good representative of such a system is Virtuoso⁵, because it is widely used in practice and in many SPARQL performance evaluations. It is built on top of its own full-featured relational database and provides both a SQL and a SPARQL front-end. Since Version 7, triples are stored column-wise and indexed in a way corresponding to the two permutations PSO and POS explained above. For queries involving predicate variables, there are additional indices and more permutations can be built on demand, thus boosting efficiency on such queries at the cost of increasing the index size. Virtuoso supports full-text search via its *bif:contains* predicate, explained in Section 1. This functionality is realized via a standard inverted index and allows keywords to match literals from the knowledge base. The same approach is used by other SPARQL engines with support for keyword search, for example, in Jena (see <http://jena.apache.org/documentation/query/text-query.html>). As explained in Section 1, this does not support entity occurrences anywhere in the text like QLever’s *ql:contains-entity* predicate does.

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

2.3 Semantic Web Search

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

⁵<https://virtuoso.openlinksw.com/>

3 INDEXING

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

3.1 Knowledge-Base Index

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

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

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

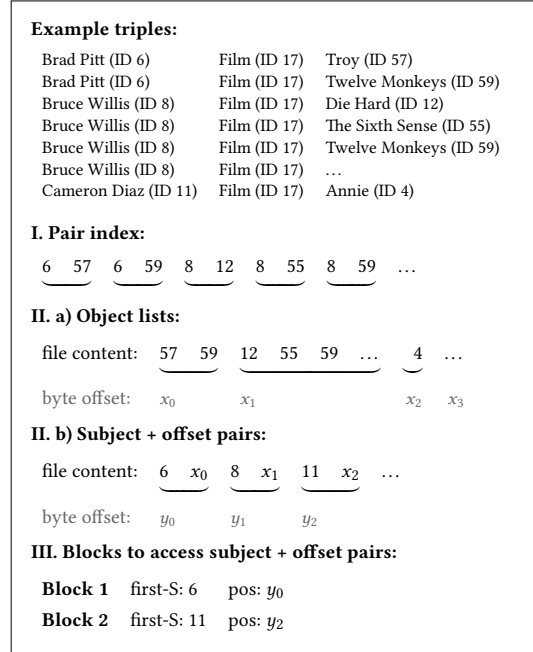


Figure 1: The components of our knowledge-base index, exemplified for a PSO permutation for a *Film* predicate.

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

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

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

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

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

We want to remark that there is a trade-off between size and speed here. First of all, we could also answer all queries with the pair index, only. It would still be decently fast, just not ideal. Secondly, we could add compression: the repeated subjects in the pair index can be gap-encoded, and so can the objects lists and subject + offset pairs. We choose to not do this for a simple reason: For large collections, the size of the text corpus usually dominates the size of the knowledge base. If disk space consumption by the KB index should be problematic for some input, it would be possible to add

compression without trouble and pay the price of slightly slower queries because of the time needed to decompress lists.

3.2 Text Index

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

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

For QLever, we also follow the main idea behind this index, but we split the list. Instead of interleaving word- and entity-postings, we keep two lists for each term (or prefix⁶). Thus, we end up with more, but shorter lists. Figure 2 illustrates these lists for a single prefix and a tiny example text excerpt.

Example text:

Text Record 21:
He<*Cristiano.Ronaldo*> scored a header in 3 consecutive games.
Text Record 23:
He<*Cristiano.Ronaldo*> headed it to Kroos<*Toni.Kroos*>.
Text Record 50:
The 2012 UCL<*UEFA.Champions.League*> final was decided by Drogba<*Didier.Drogba*>’s header.

Inverted lists for prefix head*:

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

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

II. Entity part (IDs: *Cristiano.Ronaldo*:12; *Toni.Kroos*:43; *Didier.Drogba*:15; *UEFA.Champions.League*:52):

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

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

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

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

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

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

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

3.3 Vocabulary

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

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

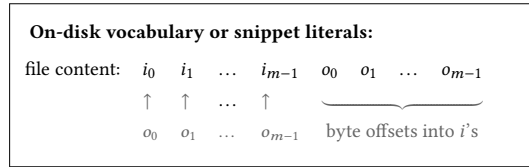


Figure 3: The first part of the file’s contents are the m items i_j written without any separator. The second part are m offsets o_j into the first part of the file, and always mark the byte offset at the end of the respective element.

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

4 QUERY PROCESSING

Our query processing has two parts: query planning and query execution. In the following, we describe the basic operations of QLever’s query execution trees and their semantics (Section 4.1),

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

our dynamic programming algorithm for query planning (Section 4.2), our heuristics for estimating result sizes and costs used during query planning (Section 4.3), and finally the algorithms to efficiently execute our basic operations (Section 4.4).

4.1 Basic Operations

The building blocks that comprise QLever’s execution trees are from a fixed set of basic operations. Many of them are standard operations and their semantics should be self-explanatory, namely: SCAN, JOIN, SORT, DISTINCT, FILTER and ORDER_BY. In the following, we describe the semantics of the less obvious operations.

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

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

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

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

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

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

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

TWO.COLUMN_JOIN is a JOIN where the joined sub-results have to match in two instead of one column.

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

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

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

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

4.2 Query Planning

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

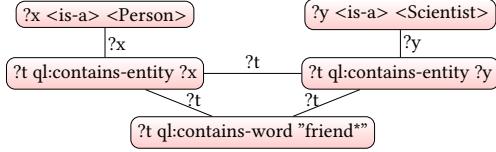


Figure 4: Graph for the first query from Section 4.1.

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



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

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

Let n be the number of nodes in the graph. We then create a DP table with n rows where the k 'th row contains all possible query execution trees for sub-queries of size k ($= k$ nodes of the

graph are included). We seed the first row with the n SCAN (or *TEXT_NO_FILTER*) operations pertaining to the nodes of the graph.

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

Whenever we merge two subtrees, a *JOIN* operation is created. Any subtree whose result is not yet sorted on the join column, is prepared by an extra *SORT* operation. There are two special cases: (1) If at least one subtree is a *TEXT_NO_FILTER* operation, we create both possible plans: a normal *JOIN* and a *TEXT_WITH_FILTER* operation.⁸ (2) If they are connected by more than one edge, we create a *TWO_COLUMN_JOIN*.

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

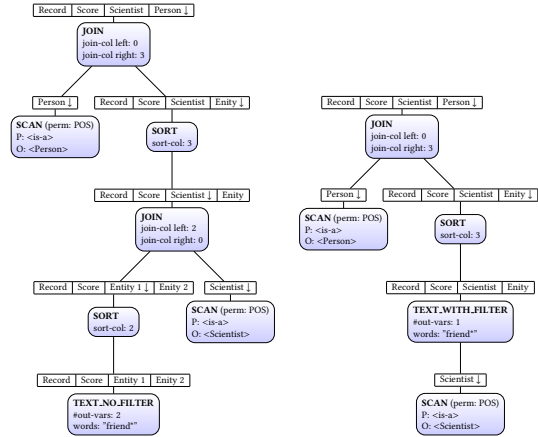


Figure 6: Two (out of many possible) example execution trees for the query from Figure 5. The right one is smaller, because of the complex *TEXT_WITH_FILTER* operation.

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

⁸When a *TEXT_WITH_FILTER* operation is created, one subtree is kept as a child and the *TEXT_NO_FILTER* operation is removed / included in the operation.

4.3 Cost, Size and Multiplicity Estimates

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

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

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

JOIN: For JOIN operations we compute the size as

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

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

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

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

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

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

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

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

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

4.4 Query execution

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

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

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

For the word with the smallest index list¹⁰, we first obtain the precomputed list of co-occurring entities (the “Entity Part” in Figure 2). For each other word, we obtain the precomputed inverted list of occurrences in the text records (the “Word Part” in Figure 2). This is a significant improvement over the query processing in [5], where all the information is read for all lists. We then intersect (using k -way “zipper”) these lists on record ID. For each pair of record and entity, we aggregate (sum up) the scores. This gives us parallel lists of *record ID*, *score*, *entity ID*, sorted by record ID.

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

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

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

⁹More sophisticated scoring schemes are possible, too.

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

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

5 EVALUATION

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

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

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

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

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

5.1 Datasets

We evaluate the systems described above on two datasets.

FreebaseEasy+Wikimedia: This dataset consist of the text of all Wikidata articles from July 2016 linked to the FreebaseEasy knowledge base [4]. FreebaseEasy is a derivation from Freebase [9] with human-readable identifiers, a reified schema (without mediator

objects), and containing all the Freebase triples with actual "knowledge" but discarding many "technical" triples and non-English literals. The example queries used throughout this paper use entity and predicate names from FreebaseEasy. The version used in our experiments has 362 million triples, the Wikidata text corpus has 3.8 billion word and 494 million entity occurrences. This dataset is similar to the Wikidata LOD dataset used in many of the INEX benchmarks (see Table 2.3 from [6]), but with about seven times more triples and a more recent version of Wikidata.

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

5.2 Queries

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

One Scan: 10 queries that can be answered with a single scan.

One Join: 10 queries that can be answered with a single join between the result of two scans.

Easy SPARQL: 10 pure SPARQL queries with small result sizes.

Complex SPARQL: 10 SPARQL queries that involve several joins, either star-shaped, paths, or mixed.

Values + Filter: 10 SPARQL queries that makes use of values (integers, floats or dates) and FILTER operations on them that compare against fixed values or each other.

Only text: 10 queries that do not involve a KB part. One or multiple words and prefixes are used to search for matching records (5 queries) or co-occurring entities (5 queries).

Is-a + Word: 10 queries for entities of a given type that co-occur with a given term.

Is-a + Prefix: 10 queries for entities of a given type that co-occur with a given prefix.

SemSearch W: 49 queries from the SemSearch'10 challenge, converted to SPARQL+Text queries without using word prefix search.

SemSearch P: 10 queries from the SemSearch'10 challenge, converted to SPARQL+Text queries using word prefix search.

Complex Mixed: 10 queries that mix several knowledge-base and text triple patterns, sometimes nested; no prefix search involved.

Very Large Text: 10 queries that return or involve a very large number of matches from the text; most use word prefixes.

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

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

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

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

5.3 Results

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

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

Freebase+ClueWeb: Table 2 reports query times of QLever for the five hardest query categories. We do not report numbers for the other systems, because they failed to index this large dataset in reasonable time on our available hardware (256 GB RAM and

¹¹The 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.

¹²20 GB for the permutations that are really needed.

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

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

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

more than enough disk space). For Virtuoso, we aborted the loading process after two weeks.

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

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

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

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

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

5.4 Variants

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

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

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

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

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

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

6 CONCLUSIONS

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

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

REFERENCES

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

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

Hannah Bast · Björn Buchhold · Elmar Haussmann

Received: date / Accepted: date

Abstract We provide a quality evaluation of KB+Text search, a deep integration of knowledge base search and standard full-text search. A knowledge base (KB) is a set of subject-predicate-object triples with a common naming scheme. The standard query language is SPARQL, where queries are essentially lists of triples with variables. KB+Text extends this search by a special *occurs-with* predicate, which can be used to express the co-occurrence of words in the text with mentions of entities from the knowledge base. Both pure KB search and standard full-text search are included as special cases.

We evaluate the result quality of KB+Text search on three different query sets. The corpus is the full version of the English Wikipedia (2.4 billion word occurrences) combined with the YAGO knowledge base (26 million triples). We provide a web application to reproduce our evaluation, which is accessible via <http://ad.informatik.uni-freiburg.de/publications>.

Keywords Knowledge Bases · Semantic Search · KB+Text Search · Quality Evaluation

1 Introduction

KB+Text search combines structured search in a knowledge base (KB) with traditional full-text search.

Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
Tel.: +49 761 203-8163
E-mail: {bast,buchhold,haussmann}@cs.uni-freiburg.de

In traditional full-text search, the data consists of text documents. The user types a (typically short) list of keywords and gets a list of documents containing some or all of these keywords, hopefully ranked by some notion of relevance to the query. For example, typing *broccoli leaves edible* in a web search engine will return lots of web pages with evidence that broccoli leaves are indeed edible.

In KB search, the data is a knowledge base, typically given as a (large) set of subject-predicate-object triples. For example, *Broccoli is-a plant* or *Broccoli native-to Europe*. These triples can be thought of to form a graph of entities (the nodes) and relations (the edges), and a language like SPARQL allows to search for subgraphs matching a given pattern. For example, find all plants that are native to Europe.

The motivation behind KB+Text search is that many queries of a more “semantic” nature require the combination of both approaches. For example, consider the query *plants with edible leaves and native to Europe*, which will be our running example in this paper. A satisfactory answer for this query requires the combination of two kinds of information:

- (1) A list of plants native to Europe; this is hard for full-text search but a showcase for knowledge base search.
- (2) For each plant the information whether its leaves are edible or not; this is easily found with a full-text search for each plant, but quite unlikely to be contained in a knowledge base.

In a previous work [4], we have developed a system with a convenient user interface to construct such

queries incrementally, with suggestions for expanding the query after each keystroke. We named the system Broccoli, after a variant of the example query above, which was our very first test query. Figure 1 shows a screenshot of Broccoli in action for this example query.

1.1 Our Contribution

The main contribution of this paper is a quality evaluation of KB+Text search on three benchmarks, including a detailed error analysis; see Section 4. On the side, we recapitulate the basics of KB+Text search (Section 2) and we provide a brief but fairly broad overview of existing quality evaluations for related kinds of “semantic search” (Section 3).

2 The Components of KB+Text Search

We briefly describe the main components of a system for KB+Text search, as far as required for understanding the remainder of this paper. KB+Text search operates on two kinds of inputs, a knowledge base and a text collection. The knowledge base consists of entities and their relations in the form of triples. The text collection consists of documents containing plain text. This input is pre-processed, indexed, and then queried as follows.

2.1 Entity Recognition

In a first step, mentions of entities from the given knowledge base are recognized in the text. Consider the following sentence:

(S) *The usable parts of rhubarb are the medicinally used roots and the edible stalks, however its leaves are toxic.*

Assuming the provided knowledge base contains the entity *Rhubarb*, the words *rhubarb* and *its* are references to it. When we index the English Wikipedia and use YAGO as a knowledge base, we make use of the fact that first occurrences of entities in Wikipedia documents are linked to their Wikipedia page that identifies a YAGO entity. Whenever a part or the full name of that entity is mentioned again in the same section of the document (for example, *Einstein* referring to *Albert Einstein*), we recognize it as that entity. We resolve references (anaphora) by assigning each occurrence of *he*, *she*, *it*, *her*, *his*, etc. to the last recognized entity

of matching gender. For text without Wikipedia annotations, state-of-the art approaches for named entity recognition and disambiguation, such as Wikify! [20], can be used instead.

2.2 Text Segmentation

The special *occurs-with* relation searches for co-occurrences of words and entities as specified by the respective arc in the query; see Figure 1 and Section 2.4 below. We identify segments in the input text to which co-occurrence should be restricted. Identifying the ideal scope of these segments is non-trivial and we experiment with three settings: (1) complete sections, (2) sentences and (3) *contexts*, which are parts of sentences that “belong together” semantically. The contexts for our example sentence (S) from above are:

(C1) *The usable parts of rhubarb are the medicinally used roots*

(C2) *The usable parts of rhubarb are the edible stalks*

(C3) *however rhubarb leaves are toxic*

Note that, because entities and references (underlined words) have been identified beforehand, no information is lost. The rationale behind contexts is to make the search more precise and “semantic”. For example, we would not want *Rhubarb* to be returned for a query for *plants with edible leaves*, since its leaves are actually toxic. Nevertheless *Rhubarb*, *edible*, and *leaves* co-occur in sentence (S) above. However, they do not co-occur in either of (C1), (C2), (C3). To compute contexts, we follow an approach for Open Information Extraction (OIE) described in [8].

2.3 Indexing

An efficient index for KB+Text search is described in [5]. This index provides exactly the support needed for the system shown in Figure 1: efficient processing of tree-shaped KB+Text queries (without variables for relations), efficient excerpt generation, and efficient search-as-you-type suggestions that enable a fully interactive incremental query construction.

2.4 Query Language

KB+Text extends ordinary KB search by the special *occurs-with* predicate. This predicate can be used to specify co-occurrence of a class (e.g., plant) or instance

The screenshot shows a search interface with the following components:



- Search Bar:** A text input field with the placeholder "type here to extend your query ...".
- Left Sidebar:**
 - Words:** A yellow box with a right-pointing arrow.
 - Classes:** A red box containing a list of classes with counts: Garden plant (24), House plant (17), and Crop (16). Below the list is a red bar indicating "1 - 3 of 28".
 - Instances:** A blue box containing a list of instances with counts: Broccoli (58), Cabbage (34), and Lettuce (23). Below the list is a blue bar indicating "1 - 3 of 421".
 - Relations:** A green box containing a list of relations with counts: occurs-with <Anything>, cultivated-in <Location> (67), and belongs-to <Plant family> (58). Below the list is a green bar indicating "1 - 3 of 7".
- Your Query:** A tree diagram showing the current query structure:
 - Root: **Plant** (with a close button 'x')
 - Child 1: **occurs-with** (with a close button 'x') leading to **edible leaves** (with a close button 'x').
 - Child 2: **native-to** (with a close button 'x') leading to **Europe** (with a close button 'x').
- Hits:** A section showing search results, with a sub-header "1 - 2 of 421".
 - Broccoli:**
 - Ontology: Broccoli
 - Broccoli: is a **plant**; native to **Europe**.
 - Document: Edible plant stems
 - The **edible** portions of **Broccoli** are the stem tissue, the flower buds, as well as the **leaves**.
 - 
 - Cabbage:**
 - Ontology: Cabbage
 - Cabbage: is a **plant**; native to **Europe**.
 - Document: Cabbage
 - The only part of the **plant** that is normally **eaten** is the **leafy** head.
 - 

Fig. 1 A screenshot of Broccoli, showing the final result for our example query. The box on the top right visualizes the current KB+Text query as a tree. The large box below shows the matching instances (of the class from the root node, plant in this case). For each instance, evidence is provided for each part of the query. In the panel on the left, instances are entities from the knowledge base, classes are groups of entities with the same object according to the *is-a* predicate, and relations are predicates. The instances are ranked by the number of pieces of evidence (only a selection of which are shown). With the search field on the top left, the query can be extended further. Each of the four boxes below the search field provide context-sensitive suggestions that depend on the current focus in the query. For the example query: suggestions for subclasses of plants, suggestions for instances of plants that lead to a hit, suggestions for relations to further refine the query. Word suggestions are displayed as soon as the user types a prefix of sufficient length. These suggestions together with the details from the hits box allow the user to incrementally construct adequate queries without prior knowledge of the knowledge base or of the full text.

(e.g., Broccoli) with an arbitrary combination of words, instances, and further sub-queries. When processing the query, this co-occurrence is restricted to the segments identified in the pre-processing step described in Section 2.2 above.

A user interface, like the one shown in Figure 1, guides the user in incrementally constructing queries from this language. In particular, a visual tree-like representation of the current query is provided after each keystroke, along with hits for that query and suggestions for further extensions or refinements.

3 Related Work

The literature on semantic search technologies is vast, and “semantic” means many different things to different researchers. A variety of different and hard-to-

compare benchmarks have therefore emerged, as well as various stand-alone evaluations of systems that perform KB+Text or variants of it.

We briefly discuss the major benchmarks from the past decade, as well as the relatively few systems that explicitly combine full-text search and knowledge base search. A comprehensive survey of the broad field of semantic search on text and/or knowledge bases is provided in [6].

3.1 TREC Entity Tracks

The goal of the TREC Entity Tracks were queries searching for lists of entities, just like in our KB+Text search. 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”. 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.

For the 2009 Entity Track [3], the underlying data was the ClueWeb09 category B collection. ClueWeb09¹ is a web corpus consisting of 1 billion web pages, of which 500 million are in English. The category B collection is a sub-collection of 50 million of the English pages. Runs with automatic and manual query construction were evaluated. This task turned out to be very hard, and the overall best system achieved an NDCG@R of 31% and a P@10 of only 24% - albeit with automatic query construction. When restricting the results to entities from Wikipedia, the best system achieved an NDCG@R of 22% and a P@10 of 45% [12]. We use the queries from this track as one of our benchmarks in Section 4 (for later tracks no Wikipedia based groundtruth is available).

For the 2010 Entity Track [1], the full English portion of the ClueWeb09 dataset was used (500 million pages). The task remained hard, with the best system achieving an NDCG@R of 37% and an R-Precision (P@10 was not reported that year) of 32% even for manually tuned queries (and 30% for automatic runs).

In 2010, an additional task was added, Entity List Completion (a similar task but with an additional set of example result entities given for each query) with BTC 2009 as the underlying dataset.² This is a dataset consisting of 1.14 billion triples crawled from the semantic web. The BTC dataset contains the complete DBpedia [18]. It turned out that the best performing approaches all boost triples from DBpedia to obtain good results. Still, working with the dataset turned out to be difficult, with the best systems achieving an R-Precision of 31% (NDCG@R was not reported).

In the 2011 track [2], another semantic web dataset was used (Sindice 2011 [13]). However, the number of participating teams was very low, and results were disappointing compared to previous years.

3.2 SemSearch Challenges

The task in the SemSearch challenges is also referred to as *ad-hoc object retrieval* [17]. The user inputs free-form keyword queries, e.g. *Apollo astronauts who walked on*

the moon or *movies starring Joe Frazier*. Results are ranked lists of entities. The benchmarks were run on BTC 2009 as a dataset.

In the 2010 challenge [17], there were 92 queries, each searching only for a single entity. The best system achieved a P@10 of 49% and a MAP of 19%.

In the 2011 challenge [10], there were 50 queries. The best system achieved a P@10 of 35% and a MAP of 28%. The 2011 queries are one of our benchmarks in Section 4.

3.3 The INEX Series

INEX (Initiative for the Evaluation of XML Retrieval) has featured many search tasks. While the focus is on XML retrieval, two tracks are remarkably similar to the benchmarks discussed before.

The Entity Ranking Track (from 2007 to 2009) and the Linked-Data Track (2012 and 2013) work on the text from Wikipedia and use intra-Wikipedia links to establish a connection between entities and an ontology or an entire knowledge base (since 2012, entities are linked to their representation in DBpedia). Queries are very similar to those of the TREC Entity Track from above: given a keyword query (describing a topic) and a category, find entities from that category relevant for that topic. However, few participants actually made use of linked data in their approaches and the results were inconclusive.

3.4 Question Answering

Question answering (QA) systems provide a functionality similar to KB+Text. The crucial difference is that questions can be asked in natural language (NL), which makes the answering part much harder. Indeed, the hardest part for most queries in the QA benchmarks is to “translate” the given NL query into a query that can be fed to the underlying search engine.

In the TREC QA tracks, which ran from 1999 to 2007, the underlying data were corpora of text documents. An overview of this long series of tracks is given in [15]. The corpora were mainly newswire documents, later also blog documents. The series started with relatively simple factoid questions, e.g. *Name a film in which Jude Law acted*, and ended with very complex queries based on sequences of related queries, including, e.g., temporal dependencies. For list questions, such as *Who are 6 actors who have played Tevye in 'Fiddler*

¹ <http://lemurproject.org/clueweb09/>

² BTC = billion triple challenge, <https://km.aifb.kit.edu/projects/btc-2009/>

on the Roof'?, which are similar to the kind we consider in this paper, the best system in 2007 achieved an F-measure of 48%.

In the QALD (Question Answering over Linked Data) series of benchmarks [19], the underlying data is again a large set of fact triples. The task is to generate the correct SPARQL query from a given NL question of varying difficulty, e.g. *Give me all female Russian astronauts* [14]. This is very different from the other benchmarks described above, where a perfect query (SPARQL or keyword) typically does not exist.

The various tracks used different sets of facts triples from DBpedia and MusicBrainz (facts about music). In the last two runs, QALD-4 [24] and QALD-5 [25], the best system achieved an an F-measure of 72% and 63%, respectively.

3.5 Systems for KB+Text and Similar Paradigms

Systems for a combined search in text documents and knowledge bases were previously proposed in [7] (ESTER), [9] (Hybrid Search), [22] (Mimir), [26] (Semplore), and [16] (Concept Search). None of these systems consider semantic context as described in Section 2.2. For all these systems, only a very limited quality evaluation has been provided.

Hybrid Search is evaluated on a very specialized corpus (18K corporate reports on jet engines by Rolls Royce). For Concept Search, a similarly small dataset of 29K documents constructed from the DMOz web directory.

For ESTER, only two simple classes of queries are evaluated: *people associated with <university>* and *list of counties in <US state>*. A precision of 37% and 67%, respectively, is reported for each class.

Semplore is evaluated on a combination of DBpedia (facts from Wikipedia) and LUBM (an ontology for the university domain). A P@10 of more than 80% is reported for 20 manually constructed queries. For many of those, the text part is simply keyword search in entity names, e.g., all awards containing the keywords *nobel prize*. Those queries then trivially have perfect precision and recall. We have only a single such query in our whole quality evaluation, all other queries combine knowledge base and full-text search in a non-trivial manner.

Mimir is only evaluated with respect to query response times and in a user study where users were asked to perform four search tasks. For these tasks, success and user satisfaction with the system were tracked.

4 Evaluation

4.1 Input Data

The text part of our data is all documents in the English Wikipedia, obtained via download.wikimedia.org in January 2013.³ Some dimensions of this collection: 40 GB XML dump, 2.4 billion word occurrences (1.6 billion without stop-words), 285 million recognized entity occurrences and 200 million sentences which we decompose into 418 million contexts.

As knowledge base we used YAGO from October 2009.⁴ We manually fixed 92 obvious mistakes in the KB (for example, the *nobel prize* was a *laureate* and hence a *person*), and added the relation *Plant native-in Location* for demonstration purposes. Altogether our variant of YAGO contains 2.6 million entities, 19,124 classes, 60 relations, and 26.6 million facts.

We build a joint index over this full text and this KB, as described in [5]. As described there, the resulting index file has a size of 14 GB with query times typically well below 100 ms [5, Table 1].

4.2 Query Benchmarks

We evaluated the quality of our KB+Text search on the dataset just described on three query benchmarks. Each benchmark consists of a set of queries, and for each query the set of relevant entities for that query from the knowledge base (YAGO in our case). Two of these query benchmarks are from past entity search competitions, described in Section 3: the Yahoo SemSearch 2011 List Search Track [23] and the TREC 2009 Entity Track [3]. The third query benchmark is based on a random selection of ten Wikipedia featured *List of ...* pages, similarly as in [7].

To allow reproducibility, we provide queries and relevance judgments as well as the possibility to evaluate (and modify) the queries against a live running system for the SemSearch List Track and the Wikipedia lists under the link provided in the abstract. The TREC Entity Track queries were used for an in-depth quality evaluation that does not allow for an easy reproduction.

³ The choice of this outdated version has no significant impact on the insights from our evaluation: the corresponding Wikipedia data from 2017 is (only) about 50% larger but otherwise has the same characteristics and would not lead to principally different results.

⁴ There is a more recent version, called YAGO2, but most of the additions from YAGO to YAGO2 (spatial and temporal information) are not interesting for our search.

Therefore we do not provide them in our reproducibility web application.

The SemSearch 2011 List Search Track consists of 50 queries asking for lists of entities in natural language, e.g. *Apollo astronauts who walked on the Moon*. The publicly available results were created by pooling the results of participating systems and are partly incomplete. Furthermore, the task used a subset of the BTC dataset (see Section 3), and some of the results referenced the same entity several times, e.g., once in DBpedia and once in OpenCyc. Therefore, we manually created a new ground truth consisting only of Wikipedia entities (compatible with our dataset). This was possible because most topics were inspired by Wikipedia lists and can be answered completely by manual investigation. Three of the topics did not contain any result entities in Wikipedia, and we ignored one additional topic because it was too controversial to answer with certainty (*books of the Jewish canon*). This leaves us with 46 topics and a total of 384 corresponding entities in our ground truth. The original relevance judgments only had 42 topics with primary results and 454 corresponding entities, including many duplicates.

The TREC 2009 Entity Track worked with the ClueWeb09 collection and consisted of 20 topics also asking for lists of entities in natural language, e.g. *Airlines that currently use Boeing 747 planes*, but in addition provided the source entity (*Boeing 747*) and the type of the target entity (*organization*). We removed all relevance judgments for pages that were not contained in the English Wikipedia; this approach was taken before in [12] as well. This leaves us with 15 topics and a total of 140 corresponding relevance judgments.

As a third benchmark we took a random selection of ten of Wikipedia’s over 2,400 `en.wikipedia.org/wiki/List_of_...` pages⁵. For example, *List of participating nations at the Winter Olympic Games*. These lists are manually created by humans, but actually they are answers to semantic queries. The lists also tend to be fairly complete, since they undergo a review process in the Wikipedia community. This makes them perfectly suited for a quality evaluation of our system. For the ground truth, we automatically extracted the list of entities from the Wikipedia list pages. This leaves us with 10 topics and a total of 2,367 corresponding entities in our ground truth.

⁵ http://en.wikipedia.org/wiki/Wikipedia:Featured_lists

For all of these tasks we manually generated KB+Text queries corresponding to the intended semantics of the original queries. We relied on the interactive query suggestions of the user interface, but did not fine-tune queries towards the results. We want to stress that our goal is not a direct comparison to systems that participated in the tasks above. For that, input, collection and relevance judgments would have to be perfectly identical. Instead, we want to evaluate whether KB+Text can provide high quality results for these tasks.

4.3 Quality Measures

Table 1 displays set-related and ranking-related measures. 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 among all relevant entities. We calculate the F-measure (F1) as the harmonic mean of precision and recall.

For our ranking-related measures, we ordered entities by the number of matching segments. 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 (AP) 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. The mean average precision (MAP) is then simply the average AP over all queries. 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 $rel(i) = 1$.

4.4 Quality Results

Table 1 evaluates our quality measures for all three benchmarks. As described in Section 2, the key com-

Table 1 Sum of false-positives and false-negatives and averages for the other measures over all SemSearch, Wikipedia list and TREC queries for the evaluated system when running on sections, sentences or contexts. The averages for F1, R-Prec, MAP, nDCG are macro-averages over all queries (that is, for example, the F1 in the first row is the average F1 of all SemSearch queries when running on sections). 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.

		#FP	#FN	Prec.	Recall	F1	R-Prec	MAP	nDCG
SemSearch	sections	44,117	92	0.06	0.78	0.09	0.32	0.42	0.44
	sentences	1,361	119	0.29	0.75	0.35	0.32	0.50	0.49
	contexts	676	139	0.39	0.67	0.43†	0.52	0.45	0.48
WP lists	sections	28,812	354	0.13	0.84	0.21	0.38	0.33	0.41
	sentences	1,758	266	0.49	0.79	0.58	0.65	0.59	0.68
	contexts	931	392	0.61	0.73	0.64*	0.70	0.57	0.69
TREC	sections	6,890	19	0.05	0.82	0.08	0.29	0.29	0.33
	sentences	392	38	0.39	0.65	0.37	0.62	0.46	0.52
	contexts	297	36	0.45	0.67	0.46*	0.62	0.46	0.55

ponent of our KB+Text search is the *occurs-with* relation, which searches for co-occurrences of the specified words / entities. We compare three segmentations for determining co-occurrence, as described in Section 2.2: *sections*, *sentences*, and semantic *contexts*.

Compared to sentences, semantic contexts decrease the (large) number of false-positives significantly for all three benchmarks.⁶ Using co-occurrence on the section level we can observe a decrease in the number of false-negatives (a lot of them due to random co-occurrence of query words in a section). However, this does not outweigh the drastic increase of the number of false-positives. Overall, semantic contexts yield the best precision on all three benchmarks, and also the best F-measure. This confirms the positive impact on the user experience that we have observed.

4.5 Error Analysis

KB+Text search, as described in Section 2 is a complex task, with many potential sources for errors. For the TREC benchmark, using contexts as segments, we manually investigated the reasons for the false-positives and false-negatives. We defined the following error categories.

⁶ For the TREC benchmark even the number of false-negatives decreases. This is because when segmenting into contexts the document parser pre-processes Wikipedia lists by appending each list item to the preceding sentence. These are the only types of contexts that cross sentence boundaries and a rare exception. For the Wikipedia list benchmark we verified that this technique does not include results from the lists from which we created the ground truth.

For false-positives:

- (FP1) a true hit was *missing* from the ground truth;
- (FP2) the context has the *wrong meaning*;⁷
- (FP3) a mistake in the *knowledge base*;
- (FP4) a mistake in the *entity recognition*;
- (FP5) a mistake by the *parser*;⁸
- (FP6) a mistake in *computing contexts*.

For false-negatives:

- (FN1) there seems to be *no evidence* for this entity in Wikipedia based on the query we used (the fact might be present but expressed using different words);
- (FN2) the query elements are *spread* over two or more sentences;
- (FN3) a mistake in the *knowledge base*;
- (FN4) a mistake in the *entity recognition*;
- (FN5) a mistake by the *parser* (analogous to FP5);
- (FN6) a mistake in *computing contexts*.

Table 2 Breakdown of all errors by category.

#FP	FP1	FP2	FP3	FP4	FP5	FP6
297	55%	11%	5%	12%	16%	1%
#FN	FN1	FN2	FN3	FN4	FN5	FN6
36	22%	6%	26%	21%	16%	8%

Table 2 provides the percentage of errors in each of these categories. The high number in FP1 is great

⁷ This means that the words occur in the context, but with a meaning different from what was intended by the query.

⁸ The sentence parses are required to compute contexts.

Table 3 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*.

	Prec.	Recall	F1	P@10	R-Prec	MAP	nDCG
TREC Entity Track, best	n/a	n/a	n/a	0.45	0.55	n/a	0.22
KB+Text, orig	0.45	0.67	0.46	0.58	0.62	0.46	0.55
KB+Text, orig + miss	0.67	0.73	0.65	0.79	0.77	0.62	0.70
KB+Text, orig + miss + corr	0.88	0.89	0.86	0.94	0.92	0.85	0.87

news for us: many entities are missing from the ground truth but were found by the system. Errors in FN1 occur when full-text search with our queries on whole Wikipedia documents does not yield hits, independent from semantic contexts. Tuning queries or adding support for synonyms can decrease this number. FP2 and FN2 comprise the most severe errors. They contain false-positives that still match all query parts in the same context but have a different meaning and false-negatives that are lost because contexts are confined to sentence boundaries. Fortunately, both numbers are quite small.

The errors in categories FP and FN 3-5 depend on implementation details and third-party components. The high number in FN3 is due to errors in the used knowledge base, YAGO. A closer inspection revealed that, although the triples in YAGO are reasonably accurate, it is vastly incomplete in many areas. For example, the *acted-in* relation contains only one actor for most movies. This could be mitigated by switching to a more comprehensive knowledge base like Freebase [11]; indeed, our latest demo of Broccoli is using Freebase instead of YAGO [4]. To mitigate the errors caused by entity recognition and anaphora resolution (FP4+FN4), a more sophisticated state-of-the-art approach is easily integrated. Parse errors are harder. The current approach for determining contexts heavily relies on the output of a state-of-the-art constituent parser. Assuming a perfect parse for every single sentence, especially those with flawed grammar, is not realistic. Still, those errors do not expose limits of KB+Text search with semantic contexts. The low number of errors due to the context computation (FP6+FN6) demonstrates that the current approach (Section 2.2) is already pretty good. Fine-tuning the way we decompose sentences might decrease this number even further.

Table 3 provides an updated evaluation, with all the errors induced by “third-party” components (namely FP and FN 3,4,5) corrected. The last row shows the

high potential of KB+Text search and motivates further work correcting the respective errors. As argued in the discussion after Table 2, many corrections are easily applied, while some of them remain hard to correct perfectly.

The first line of Table 3 shows the best results from the TREC 2009 Entity Track (TET09), when restricted to entities from the English Wikipedia; see [12, Table 10]. There are a few things to note in this comparison. First, TET09 used the ClueWeb09 collection, category B. However, that collection contains the English Wikipedia, and participants were free to restrict their search to that part only. Indeed, the best systems strongly boosted results from Wikipedia. Second, results for TET09 were not sets but ranked lists of entities, hence absolute precision and recall figures are not available. Our results are for the simplistic ranking explained above. Third, we created our queries manually, as described at the end of Section 4.2 above. However, TET09 also permitted manually constructed queries, but those results were not among the best. Fourth, the ground truth was approximated via *pooling* results from the then participating systems [3]. This is a disadvantage for systems that are evaluated later on the same ground truth [21]. Still, our quality results are better even on the original ground truth, and much better with missing entities (FP1) added.

5 Conclusions and Future Work

We have evaluated the quality of KB+Text search on three benchmarks, with very promising results. A detailed error analysis has pointed out the current weak spots: missing entities in the knowledge base, missing evidence in the full text, errors in the entity recognition, errors in the full parses of the sentences. Promising directions for future research are therefore: switch to a richer knowledge base (e.g., Freebase), switch to a

larger corpus than Wikipedia (e.g., ClueWeb), develop a more sophisticated entity recognition, try to determine semantic context without full parses.

References

1. Balog, K., Serdyukov, P., de Vries, A.P.: Overview of the TREC 2010 Entity Track. In: TREC (2010)
2. Balog, K., Serdyukov, P., de Vries, A.P.: Overview of the TREC 2011 Entity Track. In: TREC (2011)
3. Balog, K., de Vries, A.P., Serdyukov, P., Thomas, P., Westerveld, T.: Overview of the TREC 2009 Entity Track. In: TREC (2009)
4. Bast, H., Baurle, F., Buchhold, B., Hauffmann, E.: Semantic full-text search with broccoli. In: SIGIR, pp. 1265–1266. ACM (2014)
5. Bast, H., Buchhold, B.: An index for efficient semantic full-text search. In: CIKM (2013)
6. Bast, H., Buchhold, B., Haussmann, E.: Semantic search on text and knowledge bases. *Foundations and Trends in Information Retrieval* **10**(2-3), 119–271 (2016). DOI 10.1561/15000000032. URL <http://dx.doi.org/10.1561/15000000032>
7. Bast, H., Chitea, A., Suchanek, F.M., Weber, I.: Ester: efficient search on text, entities, and relations. In: SIGIR, pp. 671–678 (2007)
8. Bast, H., Haussmann, E.: Open information extraction via contextual sentence decomposition. In: ICSC (2013)
9. Bhagdev, R., Chapman, S., Ciravegna, F., Lanfranchi, V., Petrelli, D.: Hybrid search: Effectively combining keywords and semantic searches. In: ESWC, pp. 554–568 (2008)
10. Blanco, R., Halpin, H., Herzig, D.M., Mika, P., Pound, J., Thompson, H.S., Duc, T.T.: Entity search evaluation over structured web data. In: SIGIR Workshop on Entity-Oriented Search (JIWES) (2011)
11. Bollacker, K.D., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: SIGMOD, pp. 1247–1250 (2008)
12. Bron, M., Balog, K., de Rijke, M.: Ranking related entities: components and analyses. In: CIKM, pp. 1079–1088 (2010)
13. Campinas, S., Ceccarelli, D., Perry, T.E., Delbru, R., Balog, K., Tummarello, G.: The sindice-2011 dataset for entity-oriented search in the web of data. In: Workshop on Entity-Oriented Search (EOS), pp. 26–32 (2011)
14. Cimiano, P., Lopez, V., Unger, C., Cabrio, E., Ngomo, A.C.N., Walter, S.: Multilingual question answering over linked data (QALD-3): Lab overview. In: CLEF, pp. 321–332 (2013)
15. Dang, H.T., Kelly, D., Lin, J.J.: Overview of the TREC 2007 Question Answering Track. In: TREC (2007)
16. Giunchiglia, F., Kharkevich, U., Zaihrayeu, I.: Concept search. In: ESWC, pp. 429–444 (2009)
17. Halpin, H., Herzig, D.M., Mika, P., Blanco, R., Pound, J., Thompson, H.S., Tran, D.T.: Evaluating ad-hoc object retrieval. In: Workshop on Evaluation of Semantic Technologies (WEST) (2010)
18. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web* **6**(2), 167–195 (2015)
19. Lopez, V., Unger, C., Cimiano, P., Motta, E.: Evaluating question answering over linked data. *J. Web Sem.* **21**, 3–13 (2013)
20. Mihalcea, R., Csomai, A.: Wikify! Linking documents to encyclopedic knowledge. In: CIKM, pp. 233–242 (2007)
21. Sanderson, M.: Test collection based evaluation of information retrieval systems. *Foundations and Trends in Information Retrieval* **4**(4), 247–375 (2010)
22. Tablan, V., Bontcheva, K., Roberts, I., Cunningham, H.: Mimir: An open-source semantic search framework for interactive information seeking and discovery. *J. Web Sem.* **30**, 52–68 (2015)
23. Tran, T., Mika, P., Wang, H., Grobelnik, M.: SemSearch’11: the 4th Semantic Search Workshop. In: WWW (Companion Volume) (2011)
24. Unger, C., Forascu, C., López, V., Ngomo, A.N., Cabrio, E., Cimiano, P., Walter, S.: Question answering over linked data (QALD-4). In: Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15–18, 2014., pp. 1172–1180 (2014)
25. Unger, C., Forascu, C., López, V., Ngomo, A.N., Cabrio, E., Cimiano, P., Walter, S.: Question answering over linked data (QALD-5). In: Working Notes of CLEF 2015 - Conference and Labs of the Evaluation forum, Toulouse, France, September 8–11, 2015. (2015)
26. Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y., Pan, Y.: Semplere: A scalable IR approach to search the web of data. *J. Web Sem.* **7**(3), 177–188 (2009)