

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,baeurlef,buchholb,haussmae}@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, 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 (32 GB XML dump) combined with the YAGO ontology (26 million facts). We have implemented a fully-functional prototype based on our ideas, see <http://broccoli.informatik.uni-freiburg.de>.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Context Analysis and Indexing—*Indexing methods, Linguistic processing*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*Query formulation, Retrieval models, Search process*; H.5.2 [Information Interfaces and Presentation]: User Interfaces—*Theory and methods*

General Terms

Algorithms, Design, Experimentation, Human Factors, Performance

Keywords

Semantic full-text search, search as you type, Wikipedia

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’s 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 (after the entity recognition phase) 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; this will be explained

Your Query:

```

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

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

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 are updated after every keystroke. The suggestions depend on the current focus in the query, here: suggestions for subclasses of plants, suggestions for instances of plants that lead to a hit (those are exactly those from the hits box, and in the same order), suggestions for relations to further refine the query. Word suggestions make no sense at this point in the query; see Figure 2. One of the suggestions is always highlighted, in this case the *cultivated-in* relation. Pressing Return will extend the query by adding the highlighted suggestion.

in detail in Section 5.3.

Figures 1 and 2 show screenshots of our search engine in action for our example query. The figure and its caption 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 on <http://broccoli.informatik.uni-freiburg.de>.

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 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 (32 GB XML dump, 290 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 ex-

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

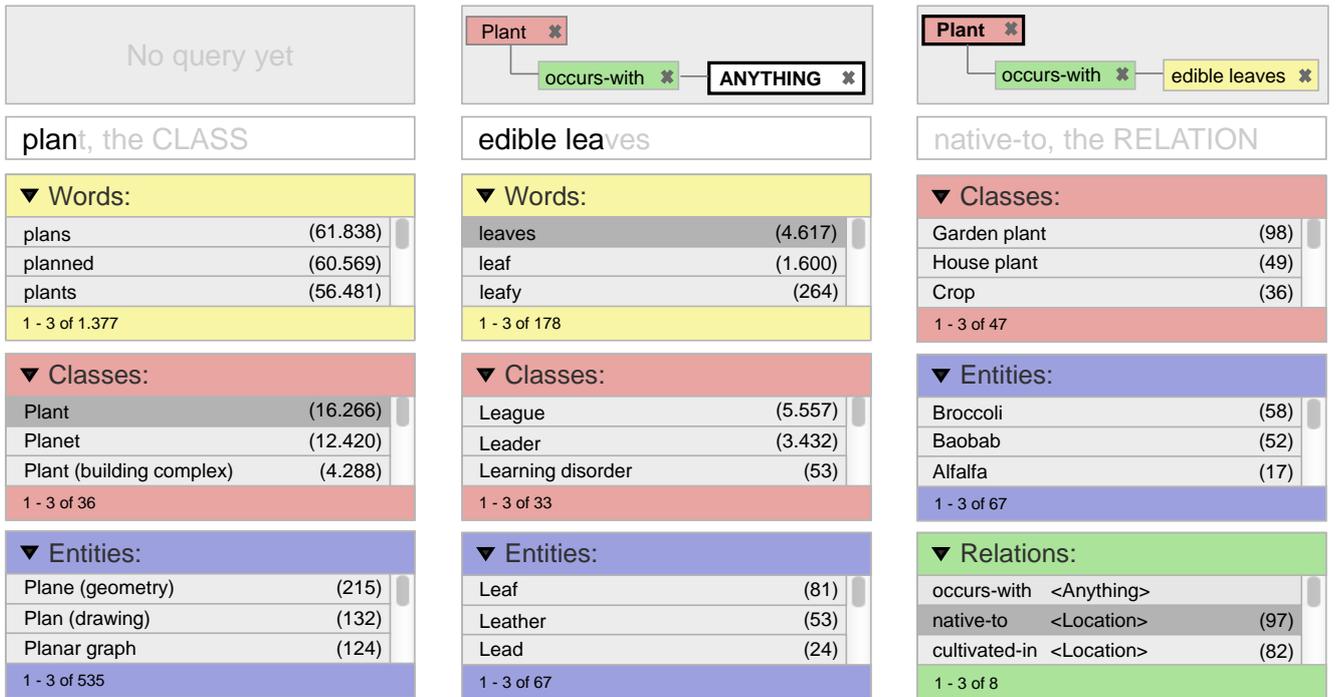


Figure 2: Snapshots of the query, search field, and suggestion boxes for three stations in the construction of our example query. Column 1: At the beginning of the query, after having typed *plan*. Column 2: After *plant* has been selected (by pressing Return) and the *occurs-with* relation has been added (by pressing Return again) and having typed *edible lea*. Column 3: After having selected *edible leaves* (by pressing Return). The focus automatically goes back to the root node, and the top relation that is not *occurs-with* is pre-selected (we could select and add another *occurs-with* if we wanted to though).

ample, and Section 6 for details.

We provide experimental results on index construction times, query times, and result quality for the English Wikipedia combined with the YAGO ontology [18]. For the quality results, we use two benchmarks: 15 queries from the TREC 2009 Entity Track (e.g., *Airlines that currently use Boeing 747 planes*), and 10 lists from Wikipedia (e.g. *List of plants with edible leaves*). In both cases, we achieve good precision, and we find many additional instances that were missing, especially from the Wikipedia lists. 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 complex 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, which we consider a very fruitful direction for future 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 [5] 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; this is explained in more detail in Section 5.1. 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.

Following Ester, various other systems offering combinations of full-text and ontology search have been proposed. Semplore [19] 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 (see Section 5.1), and no particular UI is offered. In Hybrid Search [7], 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 [13] 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. GoNTogle [12]

combines full text with annotations which are searched separately and then combined, similarly as in [7]. Queries are bags of words. There is no full ontology search and no particular natural language processing. Faceted Wikipedia Search [14] 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 [19], and query processing is DB-based and very slow, despite the relatively small amount of data. SIREN (<http://siren.sindice.com>) 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 [20] try to interpret a given keyword query semantically and translate it to a suitable SPARQL query for pure ontology search.

2.2 Systems for entity retrieval

Entity retrieval is a relatively recent 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 TREC (<http://ilps.science.uva.nl/trec-entity>). 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 benchmark collections are only a part of the web, but still very large. The ClueWeb09 collection introduced at TREC 2009 is 25 TB of text. The relative information content is low, however, 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.*² Even then the tasks remain very hard, and, for example, $P@10$ figures average only around 30% even for the best systems [11].

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. As we will see in Section 7, 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 2000 core hours on the 32 GB XML dump of the English Wikipedia, see Section 7.3. And Wikipedia’s rule of linking the first occurrence of an important entity in an article to the respective Wikipedia article helps us for a high-quality entity recognition; see Section 3.2.

Bringing Broccoli’s functionality to web search is a very reasonable next step, but out of scope for this article.

2.3 Information extraction and pure ontology search

Systems for pure 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 [16].

²In our framework these are queries with two nodes and one *occurs-with* edge.

As part of the “Semantic Web” 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. [4]). For a relatively recent survey, see [17].

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 easy to extract from existing data sources in large quantity and with high 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 the extraction of more complex information 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 question. 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. 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 which forms a tree; 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 gender³. 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 7.6 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*

³The YAGO ontology does not provide enough information to tell male from female, but we can easily tell male/female from neuter.

(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* [9] [15].

In the following, as in the example above, 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. The results from our quality evaluation in Section 7.6 (error category FN2 in Table 4) confirm 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.

In general, sub-clauses can again contain enumerations, some or all of which can again contain sub-clauses, and so on. And vice versa. In practice the nesting level is one or two for most sentences.

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.

We construct our SCI trees based on the output of a state-of-the-art constituent parser. We use SENNA [10], because of its good trade-off between parse time (around 35ms per sentence) and result quality (see Section 7.6).

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 (for example, *which* or *who*) define the

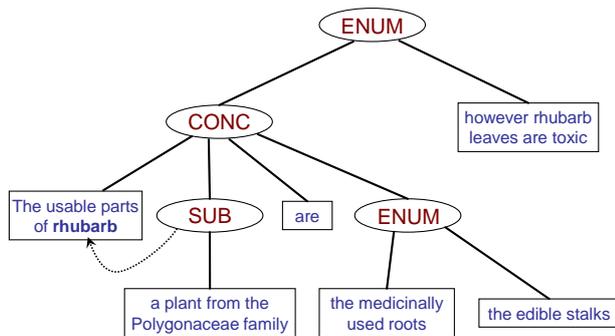


Figure 3: The SCI tree for our example sentence, after anaphora resolution. The head of the sub-clause is printed in bold.

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 (for example: *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 7.6 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.⁴

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 in regions at or near the equator.

For our definition of the answer set for a given query, we refer to Section 5.3, which describes the (recursive) algorithm for processing an arbitrary given query.

5. INDEX AND QUERY PROCESSING

It will be instructive to first consider how a standard inverted index can be used to process our semantic full-text queries and why that approach is bound to be slow (Section 5.1). We will then describe our new index (Section 5.2) and how it can be used for fast query processing (Section 5.3).

5.1 Index: straightforward approach

We first describe the straightforward realization of semantic full-text search with an inverted index. For example, this is the approach taken in [19] and [13], both using Lucene.

For each recognized entity (see Section 3.2), we simply add an item to the index representing the meant entity. To enable fast searches for classes of objects, we also need to add, for each occurrence of an entity, an index item for each class that object is in. For example, for each occurrence of the entity *Broccoli*, we would add an index item for the classes *Plant* and *Vegetable* as well.

For the English Wikipedia combined with the YAGO ontology, this simple approach will blow up the index size by a factor of 2-3, which is tolerable. What becomes a performance problem is that the inverted lists for some of the newly introduced items become very long. For example, the inverted lists for the class *Person* (which contains one item per occurrence of a person anywhere in the text) has over 50 million index items for the English Wikipedia. Note that queries for a *person* are indeed frequent, and that we indeed need one index item per occurrence (and not just one per document). Otherwise we could not specify co-occurrence in the same context, which is key for our approach. Some queries also involve the top-level class *Entity*, which contains over 165 million index items.

Moreover, the kind of queries we support here require more complex operations than just intersection on these inverted lists (like in ordinary full-text search). In [5], an addi-

⁴For queries with an instance at the root, at most one result is returned: that instance (if the rest of the query matches). These queries are still useful for browsing the evidence for that particular hit.

tional database-style join operation is implemented. But despite a highly optimized implementation, sub-second query times could be achieved there only for queries involving classes with relatively short inverted lists, like *counties* and *scientists*; see [5, Section 7.1]. As we see in the following section, our new index avoids inverted lists for instances and classes altogether.

5.2 Index: our new approach

Our new index is based on two key ideas. The first idea is taken from [6]: use inverted lists for prefixes instead of words. This enables fast query suggestions.

The second idea, which is the main idea behind our new index, is to have what we call *context lists* instead of 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, which in our current implementation is 1 for most index items, more than 1 for occurrences of an entity in its own document, and 0 for “abstract” entities like *Legume*. The fourth row contains the position of the word or entity in the respective context. This is used for proximity search and for highlighting the query elements in the result snippets. The context lists are sorted by context id, and, for equal context ids, by word/entity id, with entities coming after the words.

Relations are stored in the straightforward way, with one index list per relation. For example:

	...	#Okra	#Broccoli	#Rhubarb	...
native-to:	...	#Africa	#Europe	#Europe	...
	...	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. 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 *native-to (reversed)*.

5.3 Query processing with the new index

Recall from Section 4 that our queries are trees, where each inner node is labeled by a class or instance. However, in our internal query representation, each inner node is a

(distinct) free variable and the class/instance labels are realized via additional relational arcs of the kind $\langle var \rangle \text{ is-a } \langle class \rangle$ or $\langle var \rangle \text{ equals } \langle entity \rangle$. The former is more convenient for the user, while the latter is more convenient for the description of our query processing algorithm in the following, and also for its implementation.

The final result for a query is always a sorted list of entity ids with scores, and so are intermediate results for subtrees with a single variable as root. The intermediate result for target nodes of *occurs-with* arcs are context lists as described in Section 5.2. We process queries recursively as follows:

Ontology arcs

(QP 1) For each *ontology arc* compute the following sorted list of entities, where R denotes the relation of the arc:

(QP 1.1) Recursively compute for the target node t (which can be the root of an arbitrary query again) the result E_t , which is a sorted list of entity ids with scores.

(QP 1.2) Fetch the index list I_R for the relation R , which is sorted by target entity; see Section 5.2.

(QP 1.3) 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.

Occurs-with arcs

(QP 2) For each *occurs-with arc* compute the following sorted list of entities, where $W = \{w_1, \dots, w_k\}$ denotes the set of words or prefixes in the target node, and $Z = \{z_1, \dots, z_\ell\}$ denotes the set of entities or classes in the target node.

(QP 2.1) For each z_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 2.2) For each w_j , compute a context list C_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 .⁵ 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 C_j , and if at least one item matches append all entity index items from that context, too.

(QP 2.3) Intersect the C_1, \dots, C_k , computed in step 2.2, such that the result list C contains all index items (c, e) where c is a context id that occurs in each of the C_1, \dots, C_k , and e is an entity that occurs in context c . Since the C_i are sorted by context ids, this can be computed in time $N \cdot \log k$, where N is the total number of index items in the C_1, \dots, C_k .

(QP 2.4) 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 2.1. This can be done in time linear in $|C| + |E_1| + \dots + |E_\ell|$, by temporarily storing each E_i in a hash map.

(QP 2.5) 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.

⁵Words of length $< k$ get a context list on their own, and there are only single-word suggestions for prefixes of length $< k$.

Combining the sub-results for each arc

(QP 3) Let E_1, \dots, E_m be the entity lists computed in steps 2 (one list per ontology arc) and 3 (one list per occurs-with arc). The result list for the whole query is then simply the intersection of the E_1, \dots, E_m , where the scores of index items with the same entity id are again simply summed up.

Excerpts

Excerpts of the kind shown in Figure 1 are easily produced from the intermediate context lists computed in (QP 2.3). For each instance displayed to the user, we simply sum up the scores of the matching index items and show the ones with the highest sum.

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 introduction and screenshots (Figures 1 and 2) have already provided a foretaste of the capabilities of our user interface. Here is a list of further features. Due to the space constraint, we mention only the most important ones:

(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 constructions, 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; see Figure 2 and the explanation in the caption.

(UI 3) Visual representation of query: 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; see Figures 1 and 2.

(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.

7. EXPERIMENTS

7.1 Input data

Our text collection is the text from all documents from the English Wikipedia, downloaded from download.wikimedia.org in September 2011. Some dimensions of this collection: 32 GB XML dump, 1.1 billion word occurrences, 165 million recognized entity occurrences (see Section 3.2), and 153 million sentences which we decompose into 290 million contexts (see Section 3.3).

As ontology we use the latest version of YAGO from October 2009. We manually fixed 15 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, 17,661 classes, 23 relations, and 26 million facts.

We remark again that, in principle, our approach works for *any* given text collection, and for *any* given ontology. It is just that we currently use a relatively simplistic *entity recognition* tuned for Wikipedia; see Section 3.2. This could, however, be replaced by any other high-quality entity recognition.

7.2 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, and C++.

The full parse using SENNA (see Section 3.3) was run in parallel on 20 high-CPU instances on Amazon EC2, at a cost of around \$US 200. The remaining parts of the index construction and 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.

7.3 Pre-processing times and space usage

Basic parsing of the Wikipedia with our simple entity recognition and anaphora resolution (Section 3.2) takes around one hour. The parallelized full parse using SENNA (see above) takes around 12 hours. The SCI / SCR (Section 3.3) takes around five hours when not parallelized. Going from there, building the index lists (Section 5.2) takes another three hours.

Our index is kept in three separate files. The file for the context lists (Section 5.2, first part) has a size of 42 GB. The total number of postings is 2.8 as much as in a standard full-text index. The file for the relation lists (Section 5.2, second part) has a size of 1 GB. The file for the document excerpts has a size of 73 GB, which could easily be reduced to 51 GB by eliminating the redundant information the file currently contains. Compression, which we currently do not use anywhere, could reduce the size of these files further; see Section 8.

7.4 Query times, hits

A central aspect of Broccoli is its interactivity. 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; (Q4) class occurs-with one word; (Q5) class occurs-with two words; (Q6) ontology arc + occurs-with word; (Q7) ontology arc + occurs-with word and class; (Q8) ontology arc + occurs-with word and (class occurs-with one word).

We synthetically generated 1,000 queries for each type. For each query, we select each word or class as follows, starting from the root. Pick a random prefix of length 2, and consider Broccoli's top suggestions for the query build so far (top 30 for words, top 5 for classes). Note that each of these suggestions will give a non-empty result set. 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.

Query type	av-time	read	agg	filter	i+m+r
text-only 1	135ms	43ms	22ms	29ms	35ms
text-only 2	179ms	106ms	< 1ms	53ms	10ms
onto-only	2ms	1ms	< 1ms	< 1ms	< 1ms
onto+text 1	89ms	28ms	11ms	44ms	1ms
onto+text 2	112ms	69ms	14ms	31ms	5ms
onto+text 3	57ms	19ms	10ms	25ms	< 1ms
onto+text 4	119ms	41ms	3ms	60ms	1ms
onto+text 5	161ms	63ms	13ms	75ms	1ms

Table 1: Average query times for eight types of queries supported by Broccoli. Columns 3-6 provide a breakdown by sub-procedure; see the text.

The times reported in this table are for computing and showing the *hits*, that is, the contents of the large box on the right in Figure 1.

We observe that the average query times are around 100ms and that the bulk of the query time is spent in *reading* lists from the index. The other three columns provide the times for score *aggregation* (in QP 2.5 and QP 3 from Section 5.3), *filtering* (in QP 2.2 and QP 2.4), and *intersection, merging and ranking* of result lists (in QP 2.3 and QP 2.5).

7.5 Query times, suggestions

Table 2 provides the times for the various suggestion boxes to appear in response to a single query (recall the example in Figure 2 and see Section 6). We differentiate here between four kinds of stations in the query formulation process: (S1) type something in the beginning; (S2) type something to add an arc, refine a class, or restrict to an instance; (S3) type something to add something to the target node of an ontology arc; (S4) type something to add something to the target node of an occurs-with arc. We take the queries from classes (Q3) and (Q4) above, respectively, and type the respective element from left to right, starting with a prefix length of 3 for entities and 4 for words. Note that suggestions for more complex queries (Q5-8) would be faster not slower, because the results sets are smaller.

The table shows that only the word suggestions cost significant time, and that that time is used mainly for *reading* index lists and *filtering*, see above.

Station	prefix-len 3	prefix-len 4	prefix-len ≥ 5
beginning	< 1ms	< 1ms	< 1ms
add arc	4ms	4ms	4ms
onto node	1ms	< 1ms	< 1ms
occw node	—	93ms (r44%,f54%)	73ms (r58%,f39%)

Table 2: Average time for query suggestions for four different stations in the query formulation process, and for different prefix lengths.

7.6 Result quality

For our quality evaluation, we considered two benchmarks.

Our first benchmark are the topics and relevance judgements from the TREC 2009 Entity Track [3]. This track worked with the ClueWeb09 collection. We removed all rel-

evance judgements for pages that are not in the English Wikipedia; this approach was taken before in [8] as well. This leaves us with 15 queries with a decent number of relevance judgements.

Our second benchmark is a random selection of ten of Wikipedia’s over 70,000 manually compiled `en.wikipedia.org/wiki/List_of_...` pages: (WL1) plants with edible leaves; (WL2) drug-related deaths; (WL3) jazz musicians; (WL4) computer-animated films; (WL5) actors who played a president of the US; (WL6) nominees for Oscar for best actress; (WL7) oil-producing states; (WL8) unions associated with the AFL-CIO; (WL9) films set in New York City; (WL10) doping cases in sport.

These lists are compiled by humans, but actually they are answers to semantic queries, and therefore perfectly suited for a system like ours. Our ground truth for each list is the list of all entities on the Wikipedia page, with two kinds of entities taken out. First, entities which are not in our ontology (YAGO). Second, entities for which a very general and reasonable full-text query (for example, `<person name> jazz` for WL3) leads to *zero* hits. In total, about 30% of the entities in the 10 Wikipedia lists were *unfindable* in this way.

We first investigated the difference between using Broccoli with co-occurrence on the trivial sentence level and on the level of our contexts (see Section 3.3). Table 3 shows that by using contexts, the (large) number of false-positives decreases by a factor of about 2 at the price of an increase in the (small) number of false-negatives by about the same factor.

	TREC		Wikipedia	
	#FP	#FN	#FP	#FN
sentences	392	12	18.551	218
contexts	153	24	11.056	364

Table 3: Broccoli on sentences vs. on contexts.

We manually investigated the reasons for about 500 of these false-positives and false-negatives. We defined the following error categories. For false-positives: (FP1) the words in the context have a *different meaning* than what was intended by the query; (FP2) a true hit which was *missing* from the ground truth; (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) the query elements randomly co-occur somewhere in a paragraph, but actually there seems to be *no evidence* for this entity in the Wikipedia; (FN2) the query elements are *spread* over two or more contexts; (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*. Table 4 provides the percentage of errors in each of these categories.

The first thing that catches the eye is that over 60% of the false-positives turned out to be true hits that were missing from the ground truth, for both benchmarks. 12-14% are due to FP4: mistakes in our (simplistic) entity recognition. And 13-19% are due to FP3: the query elements do co-occur in a context but the meaning is not what we intended.

The good news about the false-negatives is the percentage for FN2. These are the results which we lose, no matter how perfect our ontology, entity recognition, and SCI+SCR

	#FP	FP1	FP2	FP3	FP4	FP5	FP6
Wiki	253	19%	61%	7%	12%	1%	0%
Trec	107	13%	60%	4%	14%	8%	1%

	#FN	FN1	FN2	FN3	FN4	FN5	FN6
Wiki	158	33%	11%	10%	14%	9%	18%
Trec	24	8%	8%	29%	17%	17%	21%

Table 4: Breakdown of errors by category for the false-positives (above) and the false-negatives (below), and for each of our two benchmarks separately.

are. For both collections it is only around 10% of the false-negatives. This suggests that context decomposition has the potential for drastically reducing the number of false-positives (see Table 3), without compromising recall much.

FN1 is due to lack of evidence in the collection and FN3 is due to ontology errors, two aspects which we cannot do much about here. FN6 means that the query elements did co-occur in a context according to our definition, but our SCI+SCR did not correctly decompose the respective sentence. So further improving our context decomposition is a worthwhile direction for future research.

Table 5 provides quality measures for our two benchmarks under three conditions: (1) with the original ground truth (*unfindable* entities already taken out, see above); (2) with the entities from FP2 (*missing*) added to this ground truth; (3) with the errors leading to FP1, FP4, FN3, FN4 *corrected*.

TREC 2009	P@10	R-Prec	MAP	P@100
original	0.51	0.60	0.34	0.46
+missing	0.67	0.68	0.42	0.63
+corrections	0.72	0.75	0.52	0.70

Wikipedia	P@10	R-Prec	MAP	P@100
original	0.62	0.42	0.28	0.46
+missing	0.81	0.62	0.44	0.68
+corrections	0.88	0.75	0.60	0.80

Table 5: Quality measures in three scenarios.

With respect to the original ground truth, our three precision-at figures are all around 0.5. This is comparable to the results reported in [8, Table 10] for the best run from the TREC 2009 Entity Track when restricted to the English Wikipedia. When adding the missing entities to the ground truth, precision increases dramatically (especially for Wikipedia). And by fixing avoidable errors that occurred to us in the query evaluation, the precision values climb to 70% and more.

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 evidence that Broccoli is indeed fast and easy to use, and that it 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. But there is much room for improvement and future work: (1) the query processing could be optimized in a number of ways; (2) clever caching strategies have the potential to further improve query times significantly; (3) improve on the entity recognition and anaphora resolution; (4) a smarter ranking; (5) evaluate Broccoli on a larger, web-like collection; (6) efficient high-quality SCI+SCR *without* the need for an expensive and error-prone full parse; (7) integrate simple inference heuristics to reduce the number of FN2-type errors; (8) a user study of our UI and the whole system.

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).

We want to thank Claudius Korzen for his valuable help in the laborious and time-consuming manual evaluation of our query results.

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, A. P. de Vries, P. Serdyukov, P. Thomas, and T. Westerveld. Overview of the TREC 2009 Entity Track. In *TREC*, 2009.
- [4] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *IJCAI*, pages 2670–2676, 2007.
- [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 I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.
- [7] 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.
- [8] M. Bron, K. Balog, and M. de Rijke. Ranking related entities: components and analyses. In *CIKM*, pages 1079–1088, 2010.
- [9] R. Chandrasekar, C. Doran, and B. Srinivas. Motivations and methods for text simplification. In *COLING*, pages 1041–1044, 1996.
- [10] R. Collobert. Deep learning for efficient discriminative parsing. *Journal of Machine Learning Research - Proceedings Track*, 15:224–232, 2011.
- [11] Y. Fang, L. Si, Z. Yu, Y. Xian, and Y. Xu. Entity retrieval with hierarchical relevance model, exploiting the structure of tables and learning homepage classifiers. In *TREC*, 2009.
- [12] G. Giannopoulos, N. Bikakis, T. Dalamagas, and T. K. Sellis. Gontogle: A tool for semantic annotation and search. In *ESWC*, pages 376–380, 2010.

- [13] F. Giunchiglia, U. Kharkevich, and I. Zaihrayeu. Concept search. In *ESWC*, pages 429–444, 2009.
- [14] 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.
- [15] B. B. Klebanov, K. Knight, and D. Marcu. Text simplification for information-seeking applications. In *CoopIS*, pages 735–747, 2004.
- [16] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
- [17] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.
- [18] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *J. Web Sem.*, 6(3):203–217, 2008.
- [19] 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.
- [20] 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.