ESTER: Efficient Search on Text, Entities, and Relations

Holger Bast   Alexandru Chitea   Fabian Suchanek   Ingmar Weber

Max-Planck-Institut für Informatik
Saarbrücken, Germany
{bast,chitea,suchanek,iweber}@mpi-inf.mpg.de

ABSTRACT

We present ESTER, a modular and highly efficient system for combined full-text and ontology search. ESTER builds on a query engine that supports two basic operations: prefix search and join. Both of these can be implemented very efficiently with a compact index, yet in combination provide powerful querying capabilities.

We show how ESTER can answer basic SPARQL graph-pattern queries on the ontology by reducing them to a small number of these two basic operations. ESTER further supports a natural blend of such semantic queries with ordinary full-text queries. Moreover, the prefix search operation allows for a fully interactive and proactive user interface, which after every keystroke suggests to the user possible semantic interpretations of his or her query, and speculatively executes the most likely of these interpretations.

As a proof of concept, we applied ESTER to the English Wikipedia, which contains about 3 million documents, combined with the recent YAGO ontology, which contains about 2.5 million facts. For a variety of complex queries, ESTER achieves worst-case query processing times of a fraction of a second, on a single machine, with an index size of about 4 GB.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; H.3.3 [**Information Search and Retrieval**]: Query formulation, Retrieval Models, Search process; H.5.2 [**User Interfaces**]: Theory and Methods

General Terms

Algorithms, Design, Experimentation, Human Factors, Performance

Keywords

Semantic Search, Ontologies, Entity Recognition, Index Building, Interactive, Proactive, Wikipedia

1. INTRODUCTION

The current generation of search engines does ranked keyword search: for a given query, they return a list of documents, ordered by relevance, which contain some or all of the query words. Ranked keyword search has been quite successful in the past, for popular queries in web search, as well as in benchmarks like those of TREC.

However, keyword search has its obvious limits, and there is no doubt that the next generation of search engines will be more "semantic" in one way or the other. For example, consider the query "which musicians are associated with The Beatles". This requires a search not for the literal *word* musician, but rather for *instances of the class* it denotes.

Already this simple query highlights two of the main challenges of semantic search: (1) obtain the necessary semantic information, in this case, identify each occurrence of a musician in the given text collection; and (2) make that information searchable in a convenient and efficient way.

Concerning (1), there are two ways to obtain semantic information. One way is to have humans annotate the text, for example, mark every occurrence of a musician with an appropriate tag. The other way is to obtain this information automatically, via natural language processing and/or machine learning techniques. For our application of ESTER to Wikipedia, we have implemented a semi-supervised method that learns from the semantic information provided by the Wikipedia links.

The focus of this paper, however, is on (2): given semantic information, make it searchable fast and conveniently. The main problem here is that standard IR data structures like the inverted index do not provide the necessary functionality. All research prototypes that we know of either use an ad-hoc extension of the inverted index or they are built on top of a general-purpose database management system. In either case, they do not scale well for retrieval tasks on large collections: they either use a lot of space, or they are slow, and sometimes both. This is discussed more in Section 1.2.

1.1 Our contribution

We present ESTER, a modular system for highly efficient combined full-text and ontology search. ESTER is built from three components: a query engine, an entity recognizer, and a user interface. An application of ESTER takes as input a corpus and an ontology. The task of the entity recognizer is to assign words or phrases in the corpus to the entities from the ontology they refer to. The query engine knows only words and documents, and supports only two basic operations on these: *prefix search* and *join*; this allows for very fast query processing and a very compact index. The main challenge in the design of ESTER was to map the knowledge from the ontology and the output of the entity recognizer to artificial words such that complex se-
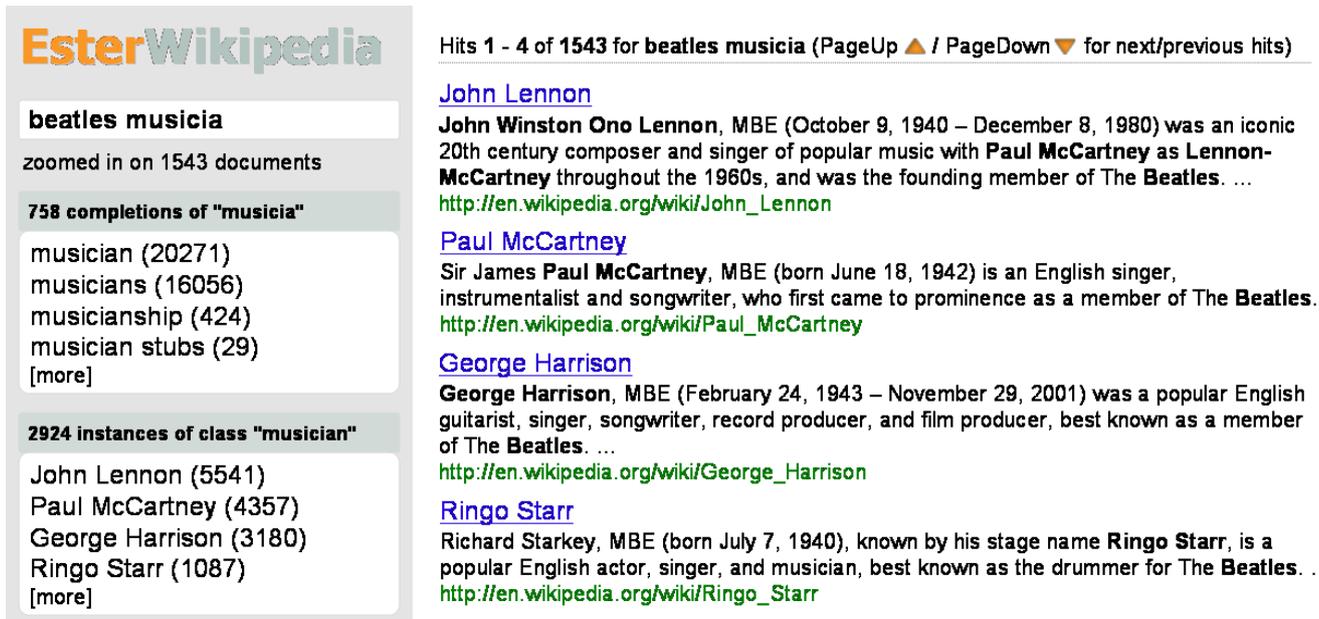
Figure 1: A screenshot of our search engine for the query `beatles musicia` searching the English Wikipedia. The list of completions and hits is updated automatically and instantly after each keystroke, hence the absence of any kind of search button. The number in parentheses after each completion is the number of hits that would be obtained for that particular completion. The upper box suggests words and phrases that start with `musicia` and that occur together with the word `beatles`. The lower box suggests *instances of musicians* that occur together with the word `beatles`. In fact, fast processing of this apparently simple query requires the whole complexity of our system in the background: prefix queries, join queries, entity recognition, and ontological knowledge; see Section 4. Our interactive and proactive (suggest-as-you-type) user interface hides this complexity from the user as much as possible. See Section 6 for other types of queries which ESTER can handle in a similar fashion.

mantic queries can be processed by mere prefix search and join operations. We show how this can be done efficiently for all basic SPARQL graph pattern queries.

As a proof of concept, we have implemented the whole system, with the query engine based on [2], an entity recognizer following insights from [6], and a user interface inspired by [2]. We applied it to the Wikipedia corpus together with the YAGO ontology [17], and conducted a variety of experiments regarding both efficiency and search quality. The key novelties of ESTER are as follows:

Scalability By building on a query engine of the kind described, ESTER can process complex semantic queries extremely fast, with a very compact index. On the Wikipedia corpus, which has about 3 million documents, together with the YAGO ontology, which has about 2.5 million facts, ESTER achieves processing times of a fraction of a second for a variety of complex queries, with an index size of just about 4 GB. This comes close to state of the art full-text search with respect to both query processing time and index size, but with much enhanced querying cababilities; see Section 7. Compared to systems with comparable querying capabilities, ESTER ist faster by up to two orders of magnitude; see Section 1.2.

Modularity Each of ESTER's three components is easily exchangeable. Any query engine can be used, as long as it can process the two basic operations: prefix search and join, required by ESTER. (For example, one might consider the method from [1] which can beat that of [2], when the data fits into main memory.) Similarly, the only requirement to

ESTER's user interface is that it translates whatever input it gets from the user to these two basic operations. Since the task of the entity recognizer is independent of the indexing and query processing, it is easily exchangeable, too.

Queries supported We show how ESTER can solve arbitrary basic SPARQL graph-pattern queries, by reducing them to the basic operations of prefix search and join. If the SPARQL query is a tree with m edges, we can show that at most $4m$ basic operations are needed. SPARQL is one of the standard query languages for ontologies, and a query is essentially a labeled graph to be matched against the ontology graph; see Section 5.

User interface ESTER has an intuitive user interface that is both interactive and proactive. For example, when a user has typed `beatles musician`, the system will give instant feedback that there is semantic information on musicians, and it will execute, in addition to an ordinary full-text query, a query searching for instances of that class (in the context of the other parts of the query), and it will show the best hits for either query. See Figure 1 for a screenshot of ESTER in action for that query.

Entity recognition For ESTER's entity recognition component, we implemented a general-purpose semi-supervised algorithm following ideas and insights from [6]. For our Wikipedia application, we took the links between Wikipedia pages as training data. We achieve a precision of about 90%, which is similar to what is reported in [6] for a collection of 264 million web pages.

1.2 Related work

There are still relatively few systems that explicitly combine full-text and ontology search. In none of the systems we know, efficiency was a primary design goal, and performance measurements are often available only as anecdotal evidence. A typical example is the recent system of [5], which supports essentially the same class of combined semantic and full-text queries as ESTER. The authors report an "average informally observed response time on a standard professional desktop computer [of] below 30 seconds" on a corpus (CNN news) with 145,316 documents and an ontology (KIM) with 465,848 facts; index size is not reported. This has to be contrasted with the subsecond query times achieved by ESTER on the 2.8 million document Wikipedia, with a provably compact index.

The powerful XQuery language can be used for the kind of queries we consider here. However, experiments (not reported in detail in this paper) with the currently fastest engine, MonetDB/XQuery [3], have shown processing times that are two to three orders of magnitude slower than what we achieve with ESTER. Another alternative is the XML fragment search of [4], which deals with a subset of XQuery and which can be used for some though not all of our queries. While most semantic search engines are built on top of a database management system, with queries being translated into SQL, the engine of [4] builds on an inverted index. Similar to ESTER, prefix search on artificial words is used, but without an efficient implementation, and neither query times nor index size are reported. In Section 7, we also briefly .

Völkel et al., in their "Semantic Wikipedia" paper [18], propose a semantic Wiki engine which makes it easy for users to add semantic information while creating or editing Wiki pages. This approach, if accepted by the community, will combine semantic information with full text information, but it will not provide a means of searching this information efficiently.

The general idea of enhancing full-text search by the addition of artificial words is (of course) not new. In the QUIQ system [11] this idea has been employed in the context of DB&IR integration. For the XML fragments search from [4] enclosing tags have been prepended to indexed words. In [14], the Wikipedia corpus has been enriched with XML tags. None of these systems uses any specialized index data structures, which does not let them scale well to large collections.

There are several works concerned with intuitive user interfaces for semantic search engines. Prominent examples are Haystack [12], Magnet [16], and the Simile tools [10]. Our proactive user interface is inspired by the full-text search autocompletion feature from [2] and the faceted-search paradigm [9]. To our knowledge, ESTER is the first system to combine semantic search with an interactive and proactive user interface.

1.3 Overview

The remainder of the paper is structured as follows. Section 2 will provide details on the query engine. Section 3 will describe how we add the ontology as artificial words to the corpus. Section 4 will describe how entity recognition gives us combined full-text and ontology search. In Section 5 we prove that ESTER can handle all basic SPARQL graph-pattern queries, and in Section 6 we describe an intuitive user interface to ESTER's low-level query engine. In Section 7, we describe our experiments with regard to both efficiency and search quality.

2. THE QUERY ENGINE

ESTER's query engine processes and produces *lists of word-in-document occurrences*, with each item consisting of a document id, a word id, a score, and a position within the document. Here is an example of such a list:

doc ids	D401	D1701	D1701	D1701	D1807
positions	5	3	12	54	9
word ids	W778	W770	W775	W774	W772
scores	0.3	0.7	0.4	0.3	0.2

For example, the first list entry says that the word with id W778 occurs in the document with id D401 at position 5 with a score of 0.3. It is important to understand that ids are assigned to words in lexicographical order, that is, lexicographically smaller words have smaller ids. The ids of all words matching a given prefix then form a *range*. We call these words the *completions* of the prefix. The query engine supports two basic operations on such lists.

Basic operation 1: prefix search Given a *sorted list D of document ids* and *a range W of word ids*, compute the list (of the kind above) of all occurrences with document id from the given set and word id from the given range.

This problem has been introduced in [2]. Note that for the special case where D is the set of all document ids, and W consists of a single word id, prefix search gives us the standard operation performed on an inverted index: get the sorted list of ids of all documents (possibly augmented by scores and positions) that contain a given word.

Here is a first, basic example of a semantic query, which can be answered by this operation alone. Assume that in our collection we have replaced each reference to *John Lennon* by the artificial word `musician:john_lennon`, and accordingly for all mentionings of a musician. We can then find all mentionings of a musician on pages mentioning the `beatles` by two prefix search queries: First, get the sorted list of all ids of documents containing the word `beatles`, by solving the prefix search query where W contains only the id of that word, and D is the set of all documents. Then perform another prefix+completion search where D is the list of these document ids and W is the list of ids of all words starting with `musician:`. This will give us the list of all mentionings of a musician in documents that also contain the word `beatles`. We will write the corresponding query as `beatles musician:*`.

For another example, assume that every musician has its own document (as is the case in Wikipedia) and that, along with the artificial word for the musician's name, we also added the birth year as, for example, `borninyear:1940`. In the same manner as for the previous example, we would then obtain the list of all musicians born in 1940 by the query `borninyear:1940 musician:*`.

Basic operation 2: join Given two occurrence lists of the kind above, compute the single list consisting of all items whose word ids occur in both lists, and sorted by document id.

For example, consider the two prefix search queries from above: the first gave us a list of all musicians occurring in

the context of the Beatles; the second gave us the list of all musicians born in 1940. Since the ids in both lists are of words of the same kind (artificial words starting with `musician:`), a join of these lists gives us the list of all musicians who are mentioned in the context of the Beatles *and* who were born in 1940. The list of document ids of the result list can be seen as "witnesses" of its individual items. Note that this example query *assembled information from different documents*; this is a kind of functionality which an ordinary inverted index cannot provide.

Realization ESTER uses the compact index from [2] to compute prefix search queries fast. The join query is a standard database operation, which can be realized in essentially two ways: by a *merge join* (sort the lists by word ids and then intersect) or by a *hash join* (compute the list of word ids that occur in both lists via hashing). For ESTER, we implemented a particularly efficient realization of a hash join which exploits that the word ranges of our queries are small. The set of word ids occurring in both lists can then be computed efficiently via two bit vectors.

3. MAPPING THE ONTOLOGY TO ARTIFICIAL WORDS

We assume the ontology to be given as a directed graph, where the nodes are labeled with entities, and the arcs are labeled with names of relations. As a minimum, we require the relations *is a* and *subclass of* with the obvious (standard) semantics that will become clear by the following examples. For our application of ESTER to Wikipedia, we picked the YAGO ontology from [17], which beyond the required *is a* and *subclass of*, contains relations such as *born in year*, *died in year*, and *located in*. YAGO was obtained by a clever combination of Wikipedia's category informations with the WordNet hierarchy [8]. The YAGO graph has about 2.5 million arcs. Example arcs, written as ordered entity-relation-entity triples are *John Lennon is a musician, John Lennon born in year 1940, musician subclass of artist*.

In the following, we describe how we cast YAGO, and similarly any other ontology which has at least the *is a* and *subclass of* relation, into *artificial words*, so that we can answer complex semantic queries efficiently using the two basic operations (prefix search and join) described in the previous section.

Ontology items as artificial words We assume that for each entity in the ontology there is a canonical document. For the Wikipedia collection and the YAGO ontology this is indeed the case; if it is not, we can simply add such canonical documents to the corpus. The construction that follows has, as a parameter, a set of *top-level categories*. The right setting of this parameter will be key to an efficient query processing. Intuitively, this set contains classes that are high up in the *subclass of* hierarchy, like *entity*, *person*, *substance*, etc.

Now consider an arc (x, r, y) from the ontology where x and y are the entities of the source and target node, respectively, and r is the relation of the arc. We then add the following artificial words to the canonical document for the entity x: At position 0, we add `<c>:<x>`, for each top-level category c of which x is an instance; at position 1, we add `<r>:<p>`, and at position p we add `entity:<y>`, where p is unique for relation r. For the special *is a* relation we further add, for each chain of triples $(x, is\ a, y_1)$,

$(y_1, subclass\ of, y_2), \ldots, (y_l, subclass\ of, z)$, the artificial word `class:<z>`.

For the three example triples from above, assuming that John Lennon is in the top-level categories entity and person, this would add (the first column gives the positions):

```
0    entity:john_lennon
0    person:john_lennon
1    is_a:2
2    class:musician
2    class:artist
1    born_in_year:3
3    entity:1940
```

Note that `entity:john_lennon` and `person:john_ lennon` are added only once, irrespectively of in how many triples the entity occurs, that all relations are added at position 1, and that the relation name contains a reference to the position of the entities from the target domain of the relation. Also note that there is no problem, if in the occurrence lists processed by ESTER several words occupy the same position in the same document.

Ontology queries Let us give a simple example for how we can make use of these artificial words. Assume we want to know the birth date of John Lennon. First, the query

```
entity:john_lennon + born_in_year:*
```

which is of the kind we have already discussed above, would give us the id of the canonical document for the entity John Lennon, as well as the (word id of the artificial word containing the) position 3. Then the query

```
entity:john_lennon + born_in_year:* ++ entity:*
```

gives us the (id of the word containing the) desired year. Here the pluses are ESTER's proximity operators: `<x> + <y>` means that `<y>` must occur at the position following `<x>`, and n pluses say that the words must have a gap of $n - 1$ positions between them. Analogously, ESTER provides the negative proximity operator `-`, with `<x> - <y>` meaning `<y> + <x>`. Note that the basic definition of prefix search given at the beginning of Section 2 can easily be extended to perform proximity search; see [2] for details.

In Section 5, we will see that with artificial words added as described we can handle arbitrary SPARQL queries. In Section 6 we will see how the artificial words together with the prefix search operation enable ESTER to free the user from having to know any special syntax or names of relation in a completely interactive and proactive way.

4. ENTITY RECOGNITION AND COMBINED QUERIES

The example queries in the previous section are purely semantic in the sense that they are operating on the ontology alone. In this section, we show how ESTER combines full-text and ontology search in an integrative manner, providing a functionality that is more than the sum of the two components.

Entity recognition We add, at the position of each occurrence of a word or phrase in the text collection that refers to an entity x from the ontology, the artificial word

```
<c>:<x>
```

for each top-level category c of which x is an instance. For example, if we take the same top-level categories as in the example from the previous subsection, then wherever *John Lennon* is mentioned (either by his full name, parts of his name, or however), we would add the artificial words

```
entity:john_lennon
person:john_lennon
```

Here we see that the set of top-level categories must not be too large: otherwise, a large number of artificial words would be added for each occurrence of an entity in the corpus, which would blow up our index beyond manageability.

Combined full-text and ontology queries   Let us give an example of which kinds of queries are possible now. Assume that we want to find all occurrences of *persons* in documents that also contain the word `beatles` (see Section 7.2 for a discussion of when and why this kind of query makes sense). Then the simple prefix query

```
beatles person:*
```

would give us the desired list. But now assume that we are looking for all occurrences of *musicians* in the set of documents matching `beatles`. Further assume that musician is not a top-level category so that we do not have artificial words of the kind `musician:<x>`. However, note that on the canonical document of each entity x that is a musician, we have the artificial word `class:musician`. Then the query

```
class:musician - is_a:* - person:*
```

will give us a list of all (ids of words containing the names of) *persons* that are musicians, where `-` is the above-mentioned negative proximity operator. A simple join of this list with the list of the previous query will now give us the desired list of all *musicians* that occur in documents which also contain the word `beatles`.

In Section 5, we show that in this fashion any basic SPARQL graph-pattern query can be processed by a combination of prefix search and join operations.

Efficiency   We have already seen that, in order to keep the index size small, we have to keep the number of top-level categories small, so that for each reference to an entity in the corpus, we add only few artificial words (one for each top-level category to which that entity belongs). The question arises, why we then not just take the top category *entity* (to which each entity belongs) as the only top-level category.

The problem is, that for a query like the one above, we would then have to execute the two queries

```
beatles entity:*
class:musician - is_a:* - entity:*
```

and join them. Now `entity:*` will have very many completions; indeed, one for every occurrence of *any* entity in the corpus. However, the prefix search queries can be processed efficiently only when the number of occurrences of words from the input word range (occurrences of words starting with `entity:` in this case) is not too large; see [2] for details. We must therefore choose the set of top-level categories such that for every sensible[1] query of the kind above, there is a

top-level category above the category we are looking for, which does not have too many completions.

Realization for Wikipedia   For our Wikipedia application, we identify occurrences of words or phrases that refer to an entity from the collection as follows. Recall our assumption that, without loss of generality, each entity in the ontology has a canonical document in the collection. Now Wikipedia has a lot of internal links, which, for selected words or phrases do exactly what we are looking for: they associate them with an entity. We use these links as training set for a simple but effective learning algorithm, which essentially follows the approach from [7].

In a nutshell, the approach proceeds in two phases: a training phase, and a disambiguation phase. In the training phase, we compute, for each word or phrase that is linked at least 10 times to a particular entity, what we call a *prototype vector*: this is a tf.idf-weighted, normalized list of all terms which occur in one of the neighbourhoods (we consider 10 words to the left and right) of the respective links. Note that one and the same word or phrase can have several such prototype vectors, one for each entity linked from some occurrence of that word or phrase in the collection.

In the second phase, we iterate over all words or phrases that have been encountered in the training phase. For each of them, we compute the similarity to each of the possible prototype vectors, by adding up those values in the prototype vector which occur in the neighbourhood of the word or phrase we are disambiguating. We then assign the meaning with the highest similarity. Similarly as in [7], we achieve a precision of around 90%, see Section 7.

5.   SPARQL QUERIES

SPARQL [19] has been proposed by the W3C as a query language for ontologies. A SPARQL query corresponding to our (purely semantic) query "musicians born in 1940" from Section 2 would be:

```
SELECT ?who WHERE {
  ?who is_a musician .
  ?who born_in_year 1940 .
}
```

These so-called *basic graph patterns* are at the core of SPARQL, and for the purpose of this section we will consider them as instances of the following *binary constraint satisfaction problem (BCSP)* [13]:

Given a directed graph G, a finite set S, for each node x a subset $S_x \subset S$ of values, and for each edge e of the graph an arbitrary relation $R_e \subseteq S \times S$. Then compute all possible assignments of values of S to the nodes of G that satisfy all relations, that is, all assignments such that for each edge e with value x assigned to its source node, and value y assigned to its target node, $(x, y) \in R_e$.

The example SPARQL query from above would correspond to a graph with three nodes x, y, and z, and two edges (x, y) and (x, z), where S_x is the set of all possible values, S_y consists of the single entity *musician*, S_z consists of the single entity *1940*, $S_{(x,y)}$ is the *is a* relation, and $S_{(x,z)}$ is the *born in year* relation.

The simplest algorithm for solving an instance of BCSP will iteratively *relax* the arcs of the given graph as follows:

---

[1]A query for all *entities* (people, substance, abstractions, etc.) associated with The Beatles will always be very expensive, for the reasons just explained. But it is not a very

sensible query precisely because it is looking for entities from a very, very general class.

for an arc (x, r, y), where the current set of values for x is X, and the current set of values for y is Y, replace X by all values which are related, with respect to r, to a value from Y, and, analogously, replace Y by all values which are related, with respect to r, to a value from X. Like this, relax the nodes in some fixed order and repeat until the sets of values do not change anymore.

There are a number of more sophisticated algorithms making use of the same basic *relax* operation [13]. It is not hard to see that in the important special case, where the query is a tree (and the vast majority of meaningful SPARQL queries are trees), it suffices to relax each arc exactly once (going from the leafs to the root). It is also not hard to see that, for ESTER, each relax operation is a matter of at most two prefix search queries and two join queries. The X and Y from above correspond to lists `<X>` and `<Y>` of occurrences in ESTER. To compute the set of all $y \in Y$ which are related, via r, to some $x \in X$, we first execute the query

```
<X> + <r> ? <c>:*
```

where c is the (top-level category encompassing the) domain of r, and `?` is to be replaced by the proximity operator pertaining to r. A join of the matching completions for the prefix `<c>:*` in the query above with the result list of `<Y>` then gives the desired subset of Y. The desired subset of X is obtained analogously. We therefore have the following theorem.

Theorem: ESTER can process an arbitrary given basic SPARQL graph-pattern query with at most $2m$ prefix search and $2m$ join queries, where m is the number of relaxations required for the solution of the corresponding binary constraints satisfaction problem. If the SPARQL query is a tree, one relaxation per arc of that tree is sufficient.

6. USER INTERFACE

Neither SPARQL nor ESTER's low-level query language (combinations of prefix searches and joins) are suitable for a front end to a search engine, where users are accustomed to extremely simple interfaces (namely, keyword search).

Inspired by the works of [2] and [9], we have therefore devised an interactive and proactive user interface for ESTER, which handles the most common types of semantic queries in a simple and intuitive manner.

We have seen a first example in Figure 1, and it would be preferable to describe the other features by appropriate screenshots, too. Due to lack of space, we describe the features in words and by example. Each of the following kinds of queries are a matter of a small SPARQL query that is a tree, and can therefore be solved efficiently by the theorem proved in the previous subsection.

Semantic completion of the last prefix This is the feature, an example of which is shown in Figure 1. At every keystroke, ESTER checks whether the last prefix of the current query string matches a class name. This is easily realized via artificial words (details omitted here). If more than one class name matches, ESTER displays a box with possible choices. For example, for the query `beatles music`, ESTER shows

```
Musical instrument (Object)
Music Genre (Relation)
Musician (Person)
```

(the class in parenthesis is the nearest top-level category containing the respective class). Clicking on one of these choices then gives a picture similar to the one of Figure 1.

Proactive display of properties of an entity If we click on one of the musicians' names in the lower box shown in Figure 1, the right panel will show documents referring to that musician prominently. On the left side, the lower box then shows a list of prominent properties of that entity according to the ontology, for example

```
John Lennon born in year 1940
John Lennon died in year 1980
John Lennon is a pacifist
```

Narrowing down a class Another frequent query is for entities from a given class which have a particular property. For example, assume we are looking for songs by German musicians. This can be formulated as the query

```
musician[german] song
```

for which ESTER displays occurrences of songs in documents that mention a musician, which in this or any other document occurs together with the word `german`. Note the subtle difference to the query

```
german musician song
```

for which ESTER displays all occurrences of songs in documents which contain the word german and mention a musician. Both kinds of queries are needed from time to time, but ESTER can be configured to translate the latter kind of query into the former automatically.

7. EXPERIMENTS

We have implemented ESTER, with the query engine described in Section 2, the entity recognizer described in Section 4, and the user interface described in Section 6. We have applied it to the Wikipedia corpus combined with the YAGO ontology. Our version of the Wikipedia corpus has 2,863,234 documents and a raw size of the corresponding xml file of 8.0 GB. Our version of the YAGO ontology graph has 984,361 nodes (entities) and 2,505,638 arcs (facts). The total number of word occurrences, including the artificial words, is 1,513,596,408, which ESTER manages to hold in an index of size 4.1 GB, including 300 MB for the (compressed) vocabulary. Our experiments were run on a machine with 16 GB of main memory, 2 dual-core AMD Opteron 2.8 GHz processors (but we used only one core at a time), operating in 32 bit mode, running Linux. The ontology-only queries were run on an Intel Pentium 4, 3 GHz, with 1 GB of main memory, running Windows. We verified that the running times on these two machines are comparable.

7.1 Efficiency

As we discussed in Sections 1.1 and 1.2, efficiency aspects have hardly been considered for other semantic search engines with similar capabilities as ESTER. Preliminary experiments (not further reported in this paper) have pointed to performance differences of up to two orders of magnitude.

For a more challenging assessment of ESTER's performance, we therefore devised the following, somewhat extreme, stress test. We constructed five query sets, three of which are purely ontological, while two others combine full-text and ontology search. For the pure ontology queries,

preliminary experiments with Jena's ARQ [15], a state of the art SPARQL engine, have again pointed to performance differences of up to two orders of magnitude; we instead let ESTER compete with the highly tuned (ontology-only) search engine that comes with YAGO [17]. For the combined queries, we compare ESTER to a state of the art engine for full-text search; for this we took our own implementation which we already used as a baseline in [2].

This comparison is extremely unfair because both the ontology-only and the full-text only system are highly tuned towards their specific task, and cannot be used for the respective other task. Moreover, YAGO's ontology-only search is realized via a database management system with one large facts table, with indexes built over each possible attribute, which means that it has the set of answers for all basic fact queries precomputed. The space requirement is accordingly high: roughly 3 GB. This has to be compared with the about 4 GB which ESTER requires for the whole full-text + ontology index. A version of ESTER built for ontology search alone has an index size of only about 100 MB.

Queries We considered the following five queries sets. Note that queries from the first three sets can be answered from the ontology alone, and do not need the full-text search capability of ESTER. For our experiments, we always used the complete index though.

Simple ontology queries: These ask for a list of triples from the ontology. Namely, for 1000 persons from the ontology, we ask for their birth year.

Advanced ontology queries: These queries require following paths in the ontology graph. Namely, for 100 relatively general classes, like biologists, social scientists, etc., we ask for all entities from that class.

Hard ontology queries: These queries require the combination of several facts from the ontology. Namely, for 1000 persons, we asked for the death dates of all persons that were born in the same year as the given person.

Combined full-text + ontology queries, Easy: These queries require the combination of full-text and ontology search as described in Section 3. For the easy set, we asked 50 queries for all counties of a given US state. The counties class is in our frontier set, so these queries can be processed via prefix search queries alone, e.g., `alabama counties:*`.

Combined full-text + ontology queries, Hard: These 50 queries ask for all computer scientists of a given nationality. Since the computer scientist class is not in our frontier set, but only the more general scientists class, these queries require more expensive prefix search queries as well as a join; see Section 4.

Results Table 1 summarizes the results of our efficiency experiments. Given the unfairness of the comparison, discussed above, the fact that ESTER's query processing times are comparable to the repective specialized baselines is a strong result in favour of ESTER.

For the full-text only queries, we simply replaced all entity prefixes by corresponding index words, e.g., `alabama county` or `german computer scientist`. Note that these queries hardly retrieve any relevant documents. We provide these figures merely to show that unlike the sytems discussed in Section 1.2, ESTER manages to stay in the same order of magnitude as state of the art full-text search. Also note

| | ESTER | | Onto-only or Full-text only | |
	avg.	max.	avg.	max.
Onto Simple	2 ms	5 ms	3 ms	20 ms
Onto Advanced	9 ms	31 ms	3 ms	794 ms
Onto Hard	64 ms	208 ms	78 ms	550 ms
Onto + Text Easy	224 ms	772 ms	90 ms	498 ms
Onto + Text Hard	279 ms	502 ms	44 ms	85 ms

Table 1: Query processing times on five query sets

that for the full-text only search, the hard query set can be processed faster than the easy set because there are more occurrences of the word `county` than of the word `scientist` in the Wikipedia. Finally, note that ESTER's query engine manages to keep also worst-case (max.) query processing times low; this is especially important for the interactive, suggest-as-you-type user interface.

7.2 Search result quality

The quality of the search results provided by ESTER, or any other semantic search engine of a similar kind, depends on three main factors: (1) the quality of the ontology; (2) the quality of the entity recognizer; and (3) the principal ability of the combined full-text and ontology search to provide interesting results.

Quality of the ontology ESTER employs an existing ontology, namely YAGO from [17]. In that paper the authors estimate, by extrapolation from human assessment on a sample, that 95% of YAGO's facts are correct

Quality of the entity recognizer Table 2 shows the quality of our entity recognizer, described in Section 4. For this assessment we held out 10% of the words or phrases for which the corresponding entities are known (because they link to some Wikipedia page), and measured the percentage of entities recognized correctly (precision). We compare our implementation (ESTER) against two simple baselines: the naive scheme that assigns every word to the most common sense that has been encountered for that word in the training set (TOP), and the scheme that assigns every word to a random sense (RANDOM).

Scheme	all words	2 senses	3 senses	$\geq$ 4 senses
ESTER	93.4%	88.2%	84.4%	80.3%
TOP	91.9%	83.5%	77.2%	77.6%
RANDOM	71.5%	50.2%	33.4%	14.0%

Table 2: The precision of ESTER's entity recognizer

Combined ontology and full-text queries quality Since there are no benchmarks for combined full-text and ontology queries on the Wikipedia, we came up with the following two *generic* (as opposed to hand-crafted) query sets:

People associated with universities (PEOPLE, 100 queries): We took the first 100 lists from the Wikipedia page "Category:Lists of people by university in the United States", for

example "List of Carnegie Mellon University people". For each such list, we generated a combined ESTER query, for example,

```
carnegie + mellon + university person:*
```

and computed the percentage of relevant entities which appear in ESTER's result list (RECALL) and the percentage of relevant entities among the top 10 (P@10), considering the entities listed on the respective Wikipedia page as the relevant ones.

Interestingly, ESTER found a number of false-negatives: for example, among the top ten entities for the above query ESTER returned Andrew Carnegie, who is not listed on the respective Wikipedia page.

Counties of American states (COUNTIES, 50 queries): For the second query set, we took 50 Wikipedia lists of the form "List of counties in <US state>".

Results As shown in Table 3, ESTER achieves an almost perfect recall and a reasonable precision on both query sets. Unfortunately we could achieve these precisions only *without* the additional entities learned by our recognizer, while recall is not affected much. This may sound paradoxical at first but there is a simple explanation: the amount of information provided by the Wikipedia links (our training data) is already very complete, so that for the broad kinds of queries from our two sets more entities only mean more noise. The entity recognizer would obviously help if we had additional documents with interesting information but without human-labeled entities, for example, news articles. This issue requires further investigation.

	PEOPLE	COUNTIES
P@10	37.3%	66.5%
RECALL	89.7%	97.8%

Table 3: Precision and recall on two query sets

8. CONCLUSIONS

We have seen that ESTER, our search engine for combined full-text and ontology search, scales very well to large amounts of data. For the Wikipedia collection combined with the YAGO ontology, ESTER can process a variety of complex queries in a fraction of a second, with an index size of only about 4 GB.

Although it was not the focus of our work, we have begun to investigate search quality for the kind of queries ESTER supports. There is certainly room for improvement here, and we are looking forward to standard entity ranking benchmarks.

It would also be interesting to conduct a user study on the perceived usefulness of semantic queries in general, and the particular type of queries ESTER supports and the interactive user interface it offers, in particular.

9. ACKNOWLEDGEMENTS

10. REFERENCES

[1] H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *13th International Conference on String Processing and Information Retrieval (SPIRE'06)*, pages 150–162, 2006.

[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *29th Annual Conference on Research and Development in Information Retrieval (SIGIR'06)*, pages 364–371, 2006.

[3] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Conference on Management of Data (SIGMOD'06)*, pages 479–490, 2006.

[4] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching xml documents via xml fragments. In *26th Annual Conference on Research and Development in Information Retrieval (SIGIR'03)*, pages 151–158, 2003.

[5] P. Castells, M. Fernández, and D. Vallet. An adaptation of the vector-space model for ontology-based information retrieval. *IEEE Transactions on Knowledge and Data Enginering*, 19(02):261–272, 2007.

[6] S. Dill, 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. A case for automated large-scale semantic annotation. *Journal of Web Semantics*, 1(1):115–132, 2003.

[7] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. V. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. Semtag and seeker: bootstrapping the semantic web via automated semantic annotation. In *12th World Wide Web Conference (WWW'03)*, pages 178–186, 2003.

[8] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[9] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49, 2002.

[10] D. Huynh, S. Mazzocchi, and D. R. Karger. Piggy bank: Experience the semantic web inside your web browser. In *4th International Semantic Web Conference (ISWC'05)*, pages 413–430, 2005.

[11] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR DB system. In *19th International Conference on Data Engineering (ICDE'03)*, pages 741–743, 2003.

[12] D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A general-purpose information management tool for end users based on semistructured data. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 13–26, 2005.

[13] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[14] R. Schenkel, F. M. Suchanek, and G. Kasneci. YAWN: A semantically annotated Wikipedia XML corpus. In *12. Symposium on Database Systems for Business, Technology and the Web of the German Society for Computer science (BTW 2007)*, 2007.

[15] A. Seaborne. ARQ — a SPARQL processor for Jena, 2005. http://jena.sourceforge.net/ARQ.

[16] V. Sinha and D. R. Karger. Magnet: Supporting navigation in semistructured data environments. In *Conference on Management of Data (SIGMOD'05)*, pages 97–106, 2005.

[17] F. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge. In *16th World Wide Web Conference (WWW'07)*, 2007. To appear.

[18] M. Völkel, M. Krötzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic wikipedia. In *15th World Wide Web Conference (WWW'06)*, pages 585–594, 2006.

[19] World Wide Web Consortium (W3C). The SPARQL query language, 2005. http://www.w3.org/TR/rdf-sparql-query.