

An Index for Efficient Semantic Full-Text Search

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

ABSTRACT

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

Categories and Subject Descriptors

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

Keywords

Semantic full-text search; Indexing; Query processing

1. INTRODUCTION

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

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

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

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

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

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

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

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2263-8/13/10 ...\$15.00.

<http://dx.doi.org/10.1145/2505515.2505689>.

▶ Words

▼ Classes:

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

1 - 3 of 28

▼ Instances:

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

1 - 3 of 3

▼ Relations:

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

1 - 3 of 7

Your Query:

Astronaut

- occurs-with walk* moon
- born-on-date <= 1930

Hits: 1 - 2 of 3


Neil Armstrong

Ontology: Neil Armstrong

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

Document: Kevin Foster (Entertainer)

Foster commented: "Now I know how **Neil Armstrong** felt when he **walked on the moon.**"



Buzz Aldrin

Ontology: Buzz Aldrin

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

Document: Upper Montclair, New Jersey

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




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

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

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

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

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

2. RELATED WORK

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

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

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

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

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

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

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

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

3. QUERY LANGUAGE AND FEATURES

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

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

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

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

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

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

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

¹<http://mg4j.di.unimi.it/>

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

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

4. LIMITATIONS OF USING KNOWN INDEX DATA STRUCTURES

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

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

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

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

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

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

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

5. THE INDEX

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

5.1 Context lists

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

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

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

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

⁴<http://dbpedia.neofonie.de/browse/>

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

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

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

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

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

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

5.2 Compression

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

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

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

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

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

5.3 Relations

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

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

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

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

Values

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

Additional Features

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

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

6. QUERY PROCESSING

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

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

6.1 Caching

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

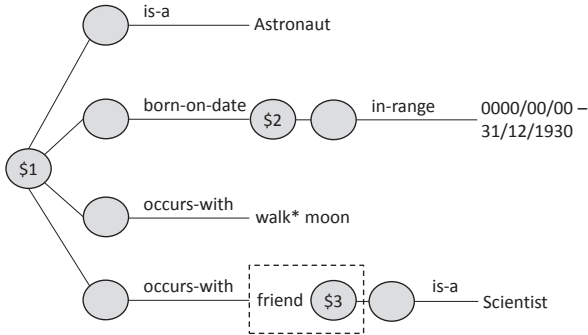


Figure 2: Query processing of an extended version of our example query. Nodes in this tree represent spots where subtree results can be computed independently and cached.

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

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

6.2 Result Computation

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

Variable nodes

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

Ontology arcs

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

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

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

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

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

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

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

Occurs-with arcs

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

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

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

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

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

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

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

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

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

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

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

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

Has-occurrence-of arcs

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

Occurs-in arcs

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

In-Range arcs

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

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

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

6.3 Excerpts

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 Suggestions

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

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

Word suggestions

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

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

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

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

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

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

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

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

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

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

Managing synonyms

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

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

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

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

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

Instance, Relation and Class suggestions

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

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

7. EXPERIMENTS

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

7.1 Experimental Setup

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

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

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

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

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

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

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

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

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

7.2 Comparative evaluation

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

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

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

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

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

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

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

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

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

7.3 Full Queries with Excerpts: Breakdown

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

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

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

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

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

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

7.4 Suggestions

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

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

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

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

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

8. CONCLUSIONS AND FUTURE WORK

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

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

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

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

9. REFERENCES

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