Annotated Search and Element Retrieval

Hugo Zaragoza[1] and Michael Matthews[1]and Roi Blanco[1] and Jordi Atserias[1]

Yahoo! Research, Barcelona (Spain)

Abstract. Despite the great interest in different forms of textual annotation (named entity extraction, semantic tagging, syntactic and semantic parsing, etc.), there is still no consensus about which *search tasks* can be improved with such annotations, and what search algorithms are required to implement efficient engines to solve these tasks. We define formally two retrieval tasks in annotated collections: annotated retrieval and element retrieval. We discuss their differences and describe efficient indexing structures, and how they can be implemented in Lucene and MG4J, two open source retrieval engines. Finally, we give a technical overview of two element retrieval use cases.

1 Introduction

There has been a great deal of work in natural language processing (NLP) over the past several decades. While there has been hope that this work will translate into better search, this still has not been clearly realized. Most NLP processing involves adding annotations to documents that help to clarify the documents intended meaning. Named-entity recognition (NER) is one of the most studied NLP techniques and one that has significant potential to improve search applications. The following is just a sample of examples queries: 1) *"[George Bush, PERSON]"*, 2) *"[George Bush, PERSON] born [DATE]"*, 3) *"IBM ceo "[PERSON]"*. Like traditional *ad-hoc* queries in Information Retrieval, these queries should return a set of document or passages. A different type of query aims at retrieving and ranking the entities themselves (instead of the documents). For example, in tasks such as person finding, entity ranking, or queries such as: 4) *"find people related to George Bush"* or 5) *find institutions (or dates, or emotionally charged verbs) related to the query "impact of terrorism on public opinion in Great Britain"*.

There are many potential search tasks over annotated collections, but they often require specialized indexes and algorithms. This makes it hard to establish a research agenda for annotated search. But as more and more examples appear (publications, applications) some trends emerge and we can hope to establish common frameworks and technologies. In this paper we propose a division into two families of *annotated search tasks*:

Annotated Retrieval: retrieval of documents relevant to an annotated ad-hoc query.

Element Retrieval: retrieval of elements (e.g. entities, not documents) relevant to an ad-hoc query.

Annotated Retrieval is the standard ad-hoc document retrieval task, with the added difficulty of coping with annotated collections and queries (examples 1-3 above). Element retrieval is significantly different from document retrieval, because what is returned is not a ranked list of documents, but rather, a ranked list of elements such as entities (examples 4-5 above).

In the remaining of the paper we discuss both tasks, with an emphasis on the later. We describe efficient indexing mechanisms required by them (Sections 2 and 3) and describe several use cases.

1.1 Related Work

With the incredible growth of the Internet, there has been a great deal of work on efficiently searching large collections [14,16] including a number of open source search implementations such as Lucene [19] and MG4J [20,6]. In the bulk of this work, a document (or passage) is represented by tokens, possibly with some simple stemming operations, but without any further linguistic annotations. There has been a parallel effort to move beyond keyword representation to capture the meaning of documents. The semantic web has long promised delivering content in machine understand form, typically based in RDF [13]. However, the bulk of information today is still in the form of free text which has lead to a surge of research into natural language processing and an increasing number of annotation frameworks such as GATE [8] and UIMA [10] which allow for large documentation annotation projects. The benefit of the resulting annotations to search applications has been shown in such areas as biotechnology, legal document retrieval, etc. Entity ranking has also been gaining interest in different forms: expert search, people search, and other forms of entity ranking have been addressed by the major evaluation campaigns (such as INEX [1] and TREC [4]). Some academic and commercial search applications have appeared on different entity search tasks, such as Beagle++ [7] which provides a semantic-based desktop search utility. However, in our opinion there is still the need of a clear framework for research in annotation retrieval and entity ranking; Rode [12] provides an early attempt on entity ranking, very much in the lines of our present work.

Several open source search engines have tackled the technical problems of implementing annotated search engines, mainly (to our knowledge): PF-Tijah [9] a native XML search-engine, Solr [22] and LuceneSail [11], which combines the keyword search capabilities of Lucene with structured data stores, and MG4J v3.0 and Archive4J [17,20] which implement parallel indexes and can be combined to build efficient element indices.

1.2 Notation

There is no standard notation for retrieval on annotated collections, and different authors have used terms to mean slightly different things (terms like entity, forward index, etc). For this reason we are forced to introduce our own term definitions and notations. We will discuss annotations and elements (entities, in a very general sense); relations are outside the scope of this paper.

A *passage* $s_i \in \mathcal{S}$ is a *sequence of tokens* plus context: $s_i = \big(w_{i1}, w_{i2}, ..., w_{i|s_i|}, \Omega\big)$. Passages may be sentences, paragraphs, documents, sliding windows of text, but they need to be uniquely identified textual units (for now we do not allow any structure on the units). Tokens will typically correspond to words, but language tokenisation is a difficult business and outside the scope of this paper. We will assume that there is some *tokeniser* that has taken raw text and segmented it into tokens. Passages may contain some context information (Ω) such as the title of the document, a surrounding window of text, external meta-data, etc.

A *token annotation* is the tuple $a = (i, \alpha, \beta, t)$ where i is the identifier of the passage annotated, α and β are the beginning and ending positions of the annotation respectively, and t is the type of the annotation. An *element* is a tuple $e = (s, t)$ where s is its value (a sequence of tokens) and t is its type. Unlike annotations, elements are not tied to a particular passage. We call $\mathcal{E}$ the set of all elements.

Annotations can be obtained by automatic or manual extraction and can be encoded in many different ways, depending on their intended usage, their sizes, types, etc. The following sections will discuss this issue.

We also need some notation to express search queries. Given the collection of passages $\mathcal{S}$, a query is defined as a function that maps $\mathcal{S}$ to a ranked subset S_q. We will use the following query operators (which can be combined into more complex queries): conjunction "$a\,b$" retrieves passages with tokens a and b, disjunction "$a|b$" retrieves passages with tokens a or b, field restriction $A : a$ retrieves passages with token a in field A, position restriction "$[a, b] \sim n$" retrieves passages where token a is followed by token b in a window of at most n tokens, and alignment "$a \wedge b$" retrieves passages where token a is aligned with (in the same position as) token b.

2 Annotated Passage Retrieval

Inverted indices allow fast and efficient retrieval of passages for token queries allowing some operators (such as disjunctive and conjunctive queries, token prefixes, constraints on match distances, etc). It is possible to modify these indices to allow similar query operators on an annotated collection.

In its basic form, an inverted index contains one *postings list* for every token present in the collection. Typically, postings lists may contain information arranged into different levels of granularity, from the presence/absence of tokens to their frequency or retrieval weight, their position, field information, typography, etc. It is possible to modify inverted indices slightly to perform retrieval and ranking over annotated collections. One approach is to implement several *parallel indexes* on the collection, by allowing several tokens in different fields to share the same position: the main index stores the positions of text tokens, and each additional index stores the annotations of the different types. At a low-level this implies building a different index for each type we want to include in a query (tokens, entity types, and so on), and implementing a fast alignment operator.

For instance, querying for the element e_1 *=[apple, SUBSTANCE]* would require a searching for the text-token *apple* in the same position as the type-token *SUBSTANCE.* The query for the element $e_1$could be translated then into *"apple:TOKENS ^ SUBSTANCE:TYPES".* A passage would satisfy this query if both *apple* and *SUBSTANCE* are found in the same passage at the same position. Since the alignment operator preserves the semantics of the other query operators, the result is a very powerful query language over annotated collections. Alignment can be used in combination with phrasal queries, negation, positional restrictions, etc. For example, we could query for *"pizza or pasta in New York"* as *"[(pizza|pasta) [New^CITY, York^CITY]~1]]~10".*

Note that in the examples above we did not enforce the limits of the elements: our queries would incorrectly match longer elements (such as *[apple pie, SUBSTANCE]*) or sequences of short ones (such as *"[New, CITY] [York, CITY]"*). This can be solved by encoding the limits into the types. In particular, we concatenate to the type a character encoding the bracketing (e.g. beginning (B), ending (E), continuing (I) or a single token (U)). This way we can query for *"apple^SUBSTANCE_ U"* and *"[New^CITY_B, York^CITY_E]~1".*

There is a special type of annotated query that is common in applications and that does not require a full-blown parallel index: a *type restriction.* Here, we want to restrict our search to passages that contain a particular annotation type (e.g. *"Einstein DATE"*). We are not concerned with the position of the type annotation in the text, or even with the actual value of the annotation. This type of filtering is easily implemented using fields (without positions) where we simply store the types present in the passage.

Parallel inverted indices can be easily implement in traditional search engines with positional indices, and for this reason they have been used extensively although rarely discussed in the academic literature. In the open domain, parallel indices are implemented in MG4J and can be implemented easily in other frameworks such as Lucene. For example, in Lucene, one can obtain a parallel index by writing a document reader that does not increment the position (setting *PositionIncrement* to 0) when types are encountered. Although Lucene does not provide specifically an alignment operator, one can obtain one by forcing a distance of zero between the token and the type. In MG4J parallel indices can be obtained naturally by indexing different fields and *aligning* them at query time with the alignment operator. Another example of a parallel index discussed in the literature is the *colored index* used by Attardi's IXE system [3].

3 Element Retrieval

In applications such as element ranking and faceted search we are required to provide a ranked list of elements (not documents) that are relevant to the query in some sense. This requires more analysis in the corpus than the standard retrieval task: after solving the query and obtaining a ranked list of passages (S_q), we need to find which elements appear in those documents (C_q), a potentially expensive operation.

The most straightforward method would be, for each returned passage, to load the full original annotated passage and simply traverse the passage counting the annotations. The performance would be highly dependent on the efficiency with which the passage annotations could be loaded given the passage identifier. If the original documents are stored as individual files in a file system or as records in a database system, it may be possible to retrieve 10 or even 100 passages, but retrieving even 1000 passages will not be feasible for a real time search application. It is clear that a more efficient data structure must be used to perform the counts.

If we ignore the positions of annotations (α and β) the relationship between passages and element instances can be represented conceptually as a graph, in which each passage $s \in \mathcal{S}$ and each element $e \in \mathcal{E}$ are nodes, and there is a directed edge from s to e if the element e is present (contained) in the passage s. We call this an *element containment graph* $\mathbf{C}$, and since it is a bipartite graph it can be represented as a $|\mathcal{S}| \times |\mathcal{E}|$ matrix, where C_{ij} is the strength of connection between passage s_i and element e_j (typically 1 if it appears, 0 otherwise).

At query time, we execute the standard query and obtain the set of passages of interest (S_q, or perhaps only the top-k scores in this set). Then, for every passage of interest we query C to obtain the elements contained in the passage. Doing so, we obtain the subgraph $C_q \subseteq C$ which contains information useful for ranking, such as the frequency of an entity in the result set.

Representing C as a graph allows us to use tools from graph theory and linear algebra to further understand and manipulate elements. For example, we see that the passage frequency of an element is equal to its degree in the graph, and this can be extended to the weighted degree, which takes into account the strength of the connection of each of its instances. Furthermore, $\mathbf{C}\mathbf{C}^T$ gives us a passage similarity (equal to the number of element co-occurrences if $\mathbf{C}$ is binary), and $\mathbf{C}^T\mathbf{C}$ an entity similarity (number of passage co-occurrences if $\mathbf{C}$ is binary). We can further use $\mathbf{C}$ to define graph centrality algorithms such as HITS or PageRank, or to define several types of random walks [12].

Representing $\mathbf{C}$ as a bipartite graph also allows us to use existing graph compression and querying algorithms, such as WebGraph [5]. The actual values of the entities and passages (e.g. its string values and other meta-data) are typically stored separately in *dictionaries*, for example as alphabetically ordered files (or further compressed as front-encoded lists for example). This makes it possible to keep in memory element containment graphs of millions of nodes, and query them many thousands of times per second.

Like any sparse matrix, $\mathbf{C}$ can also be represented as an index either in row (passage) order or column (element) order. A row can be seen as a description of the elements contained in a passage: $s_i : \{e_{i1}, ..., e_{in}\}$, whereas a column is a description of the passages in which an element is present $e_j : \{s_{j1}, ..., s_{jm}\}$. Both these representations lead to efficient indexes which allow querying for passages (in the first case) and for elements (in the second case). If instead of elements we were considering textual terms, then the column vectors would be similar to an *inverted index*. Pushing the analogy, some people refer to row or passage order indices as *forward* indices.

There exist efficient algorithms to store forward indices. Archive4J [17] was specifically designed for this purpose, and implements a data structure called an *archive*, which builds *a direct file* of a document collection allowing retrieval of data from a single document, specifically its length in words and which terms occur in the document with their respective term frequencies. The tool provides random access to documents while being able to obtain high compression ratios. By indexing elements instead of terms, we can obtain a forward index (a row of $\mathbf{C}$). Solr, a search engine built using the Lucene libraries, has a similar structure referred to as an *UnInvertedIndex*. Both structures are useful to rapidly determine (and possibly count) which elements are present in $\mathcal{S}_q$, the (possibly very large) set of passages returned by the query.

The following subsections describe the technical details of two applications that utilize many of these techniques.

3.1 Correlator

Correlator [18,15] is an entity retrieval demo on the English Wikipedia. It allows users to search for entities (elements of certain types) related to a free query. The collection in Correlator consists of 2,276,293 English Wikipedia entries (roughly one billion words). This collection was pre-processed with a set of linguistic tools (see [2] for an explanation) to obtain annotations following the Wall Street Journal BBN Entity Types from LDC[1]. This lead to 26,110,586 unique elements of 105 types.

In order to implement element retrieval in Correlator, we use several of the data structures discussed above. First, a *passage* is defined as a single sentence (an automatic sentence splitter was used for this, leading to 62,614,788 sentences). The context of a passage is defined as the Wikipedia entry title, plus the two sentences immediately before and after the sentence indexed. This content is indexed in a separate field to the sentence (which forms the body of the retrieval unit). Furthermore, in order to implement filtering by type, we index with each

[1] Linguistic Data Consortium: LDC2005T33

sentence a sequence of type tokens which indicate which types are present in the sentence. This type of index allows us to find passages (sentences) relevant to a query, and to filter them by type. To implement entity retrieval, we also construct the element containment graph using the top 1000 results. This graph contains one node per sentence, one node per element and one edge per annotation; the size of the resulting graph, once compressed using WebGraph, is 1.8G bytes.

3.2 Question eXplorer

Question eXplorer (QX) [21] is a browsing interface that demonstrates the power of combining linguistic parsing and fast forward indices. Every time the user types a query, besides doing the traditional retrieval, it computes statistics over all the elements present in the result set (e.g. in the query element containment graph C_q). It uses these statistics to build lists of the most frequent elements of every type and proposes them to the users for query completion. The types of elements used are very specific: noun phrases (NPs), verbs (Vs), noun modifiers (K), verb modifiers (M) and numbers (Q). In order to extract the most interesting elements in a passage, we pre-process the passage using a linguistic parser (as described in [2]) and select the elements closest to the root of the parsing tree. Furthermore, for noun phrases with more than one token, we created elements for the phrase and for sub-phrases containing the head (e.g. we created the element [“cheap computer screen”,NP], but also [“compute screen”,NP] and [“screen”,NP]).

In order to implement this efficiently we need to *rank* all the elements present in a result set. This problem is similar to the entity ranking problem, but with some differences. First, the number of entities is potentially very large: every verb and noun-phrase, and many sub-strings of these. Second, the number of types is very small (only two). Third, we want to count all the elements in the returned passages (or as many as possible), not just the top-k. In order to implement this, we encoded the element containment graph as a fast passage-element index (using the Archive4J libraries [17]). For a collection of 4,483,032 questions (roughly 48M tokens) we extracted 56,280,105 elements (roughly 390M bytes of text). The forward and backward indices are implements using MG4J. The inverted index for searching passages (standard token inverted index) results in a size of 2,144M bytes. The element forward index (implemented using the archive4J library) has a size of 656M bytes.

4 Conclusions and Future Work

We have formally defined two retrieval tasks on annotated collections and described how these tasks have been implemented for two applications using open source tools. In the future, we plan on evaluating the performance of the techniques presented in order to provide guidelines for building annotated search applications.

References

1. Overview of the inex 2008 entity ranking track. In *INEX*, 2008.
2. J. Atserias, H. Zaragoza, M. Ciaramita, and G. Attardi. Semantically annotated snapshot of the english wikipedia. In *LREC'08*, 2008.
3. G. Attardi. IXE at the TREC 2005 Terabyte Task. In *TREC*, 2005.
4. P. Bailey, A. P. de Vries, N. Craswell, and I. Soboroff. Overview of the trec 2007 enterprise track. In *Proceedings of TREC 2007 the 16th Text REtrieval Conference*, 2007.
5. P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW 2003*, pages 595–601. ACM Press, 2003.
6. P. Boldi and S. Vigna. MG4J at TREC 2005. In *TREC 2005*, Special Publications. NIST, 2005.
7. I. Brunkhorst, P.A. Chirita, S. Costache, J. Gaugaz, E. Ioannou, T. Iofciu, E. Minack, W. Nejdl, and R. Paiu. The beagle++ toolbox: Towards an extendable desktop search architecture. In *SemDesk 2006*, volume 202, November 2006.
8. H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *ACL*, 2002.
9. R. van Os D. Hiemstra, H. Rode and J. Flokstra. PF/Tijah: text search in an XML database system. *OSIR*, pages 12–17, 2006.
10. D. Ferrucci and A. Lally. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348, 2004.
11. E. Minack, L. Sauermann, G. Grimnes, C. Fluit, and J. Broekstra. The sesame lucene sail: Rdf queries with full-text search. Technical Report 2008-1, NEPOMUK Consortium, February 2008.
12. H. Rode. *From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search*. PhD thesis, University of Twente, Enschede, The Netherlands, June 2008.
13. N. Shadbolt, Berners T. Lee, and W. Hall. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006.
14. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, CA, 1999.
15. H. Zaragoza, H. Rode, P. Mika, J. Atserias, M. Ciaramita, and G. Attardi. Ranking very many typed entities on wikipedia. In *CIKM '07*. ACM Press, 2007.
16. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38:1–56, 2006.
17. Archive4j. `http://archive4j.dsi.unimi.it`.
18. Correlator. Yahoo! SandBox. `http://sandbox.yahoo.com/Correlator`.
19. Lucene. Homepage. `http://lucene.apache.org/`.
20. MG4J: Managing gigabytes for java. Homepage. `http://mg4j.dsi.unimi.it/`.
21. Query explorer. Yahoo! SandBox (to be published). `http://sandbox.yahoo.com/qx`.
22. Solr. Homepage. `http://lucene.apache.org`.