Structured retrieval using SOLR

Leonid Boytsov

CMU-LTI-15-001

Language Technologies Institute
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
http://www.lti.cs.cmu.edu/

Abstract

This report presents SOLR Annographix: a software suite for indexing and querying annotation graphs generated by NLP pipelines. Effectiveness and efficiency of the software was evaluated in a document retrieval task for the purpose of factoid question answering (QA). To this end, we employed the AQUAINT text collection and TREC QA topics. The actual queries were generated from answer-bearing sentences using an approach of Bilotti [3]. Queries that included linguistically-motivated constraints outperformed unstructured queries: The differences were substantial and statistically significant. In addition, we experimented with queries generated from randomly selected corpus sentences: It was possible to retrieve original documents in at least 90% of cases, while spending less than 0.1 sec per query (on average). These results are encouraging. Yet, the system needs to be improved by (1) implementing annotation compression, (2) extending the query language, (3) implementing better graph matching algorithms that would be less dependent on an early termination heuristic.

1 Introduction and Motivation

A question answering (QA) system aims to satisfy a precise information need expressed in a natural language. For example, the user can type a factoid query: "What is the capital of the United States", for which there exist a unique and concise answer. Modern QA systems are extractive in nature and have limited capabilities of synthesizing answers from several sources. Thus, answering a question entails finding a segment of text: a document, a passage, or a sentence, that contains an exact answer to the given question. This segment of text is often called answer-bearing. In particular, one of the most advanced QA systems IBM Watson heavily relies on extractive retrieval-base approaches [5].

In a classic QA pipeline, the question is analyzed and subsequently represented by a set of keywords and/or phrases that are used to retrieve answer-bearing passages [18]. Afterwards, downstream components can carry out a deeper textual analysis of top-k (most highly ranked) passages and extract potential answers. This process can use sophisticated natural-language processing (NLP) to verify linguistically-motivated constraints identified during the question-parsing stage. In the question "What is the capital of the United States", the answer is a city, so that we can ignore all the sentences that do not contain city names.

Quite often, the retrieval phase in a QA system relies on bag-of-words model. However, "... there is a fundamental disconnect between the capabilities of the bag-of-words retrieval model and the retrieval needs of the QA system" [3]. In that, if linguistic constraints are not used during retrieval, good answer-bearing passages can be outranked by passages that better match question keywords.

For example, the question "When did Stribling lose to Carnera by a foul?" would be hard to answer, because in one match Carnera defeated Stribling, but in a rematch Stribling took over. In that, both matches were won by a foul (perhaps, due to a conspiracy). In the sentence "Carnera defeated Stribling", Carnera is an *agent* and Stribling is a *patient*, but in the sentence "Carnera lost to Stribling by a foul" Carnera is a *patient*. To distinguish between the two outcomes, one needs to determine thematic roles at index time (the process known as semantic role labeling), and take them into account during retrieval.

Annotations are a common formalism for encoding linguistic features such as thematic roles or parts of speech (POS) tags. Electronic annotations model real-life annotation of the text, in which the user highlights certain text spans with a marker. One common extension of this formalism introduces parent-child annotation links, which are necessary to support hierarchical annotations such as dependency trees.

Two popular generic approaches to extend bag-of-words models with structured-retrieval capabilities are: storing an annotation graph in a fast forward index and creating a parallel index [25].

In the former approach, the forward index is used to verify the constraints as a post-processing step of passage retrieval. More specifically, we first retrieve top-k documents using a bag-of-words model. Next, we obtain their annotation graphs by querying the forward index. However, due to a potential disconnect between bag-of-word queries and answer-bearing sentences, answer-bearing documents may not appear among top-k documents.

In a simple parallel index, we allow several terms to share the same position. In this fashion, we can specify term properties such as POS tags (e.g., `NN`) or named entity types (e.g., `PERSON`). Under the hood, this is often implemented by keeping separate posting lists for terms and term properties. At query time, their positions are matched by a fast alignment operator [25].

This simple model has two major drawbacks. First, this approach can be space-inefficient for annotations that span multiple tokens, because each term covered by an annotation is mirrored by some entry in a parallel posting list. It is even more wasteful, if annotations store additional attributes such as IDs (identifiers) of parent annotations, because these attributes will be stored multiple times: Once for each token covered by an annotation. Second, this model relies on the assumption that (1) all the annotations start/end on easily-identifiable token boundaries, (2) all the annotating tools are using the same tokenization algorithm. These assumptions are problematic even in English, because tokenizers differ in the way they treat possessive markers, hyphens, and punctuation characters. In the case of agglutinative and other synthetic languages, it also makes sense to annotate word morphemes rather than complete words.

The implementation of the parallel index in the Indri search engine[1] is free from these issues. In Indri, there is only one entry per annotation whose start/end positions are specified via character offsets rather than token positions. In addition, each annotation includes its own ID as well as an optional ID of the parent. Indri has a powerful query language that supports both a containment and a parent-child relational operator. Indri delivers state-of-the-art retrieval performance for both bag-of-words and structured queries.

An open-source engine Apache SOLR (built on top of Apache Lucene)[2] is a rapidly developing software package, which enjoys support from a large open-source community. However, it lacks structured-retrieval capabilities similar to those of Indri. This motivated us to create a SOLR *query plugin* to fill in the gap. The current query plugin permits specifying a maximum span length (or, alternatively, a name of the covering annotation representing, e.g., a paragraph) as well the structure of a matching annotation graph.

[1]`http://www.lemurproject.org/indri/`
[2]`http://lucene.apache.org/solr/`

This plugin is complemented by an UIMA[3] analysis engine that consumes the output of an NLP pipeline and generates intermediate files in formats that can be indexed by either SOLR or Indri. In this report, we describe the architecture of the implemented query plugin, discuss the algorithmic issues, and present results of an experimental evaluation.

2 Related Work

The idea that a syntactic structure of a question should be similar to a syntactic structure of an answer has been pursued for several decades. An early example is *Protosynthex* QA system implemented by System Development Corporation [23]. Protosynthex relied on a human-guided dependency parsing. Among recent work of interest are papers by Cui et al. [7] and by Shen et al. [22], who showed that re-ranking using dependency information and semantical role labels, respectively, can substantially improve performance of a factoid QA system. Both papers rely on a bag-of-words retrieval model and use additional syntactic information only during a post-retrieval phase.

In contrast, Bilotti et al. [4] demonstrated effectiveness of a structured retrieval approach, where linguistically-motivated constraints–defined by predicted thematic roles–were used directly during retrieval. Bilotti et al. employed structured retrieval capabilities of the search engine Indri. A similar result was obtained by Miyao et al. [17], who also claimed to have invented an efficient algorithm for structured retrieval (and referenced an unpublished report).

Unfortunately, neither the algorithm by Miyao et al. nor algorithms underlying structured retrieval in Indri appear to be described in sufficient detail. In the context of searching XML data, however, there is a large body of literature concerned with the efficiency of structured retrieval [9].

There are several search engines that have structured retrieval capabilities, including Indri. However, there appear to be no open-source implementation of a SOLR or Lucene extension that provides functionality similarity to that of Indri. Siren is a structured retrieval component built on top of Lucene [8].[4] However, it supports only the nested model, which is too restrictive for most NLP applications.

3 Architecture

3.1 Indexing Model

In SOLR, the minimum indexable unit is a document. A document has an identifier and several sections called *fields*. Each field has its own set of posting lists for terms appearing in this field. Two fields are employed by our query plugin: one field is used to keep the original text of a source document, and another field is used to store NLP annotations. The text field is configured to store both term positions and term offsets. The latter are required for efficient alignment of annotations and terms during retrieval.

[3]`https://uima.apache.org/`

[4]`https://github.com/rdelbru/SIREn`

A	capital	of	the	United	States
`DT`	`NN`	`IN`	`DT`	`NNP`	`NNP`
0–1	2–9	10–12	13–16	17–23	24–30
				`LOC`	

Table 1: Sample annotations (POS tags and the named entity) for the phrase “A capital of the United States".

Our implementation is analogous to the parallel index of the Indri search engine [3]. The annotation field stores annotation labels augmented with a tuple:

`(startOff, endOff, id, parentId),`

where `startOff` is a zero-based index of the first character covered by the annotation, `endOff` is a zero-based index of the character following the last character covered by the annotation, `id` is an annotation ID (that needs to be unique within a document), and `parentId` is an ID of the parent annotation. If no parent exists, `parentId` is set to zero.

All tuples are *sorted* first by the type of the annotation (dependencies, POS tags, etc), next in the order of non-decreasing start offset. Elements of the same type with equal start offsets are sorted in the order of non-decreasing end offset.

Consider an example where the phrase “A capital of the United States" is annotated using a POS tagger and a named entity recognizer. The result is shown in Table 1, where text tokens occupy the first row, POS tags are given in the second row, the third row indicates start/end offsets of tokens and corresponding POS tags, while the last row shows the named entity annotation spanning two terms.

Assuming that annotation IDs are assigned left-to-right, the first annotation `DT` has the ID one and the named entity has the largest ID, the annotation index will contain the following tuples:

- (0, 1, 1, 0) & (13, 16, 4, 0) for the tag `DT`;
- (2, 9, 2, 0) for the tag `NN`;
- (10, 12, 3, 0) for the tag `IN`;
- (17, 23, 5, 0) & (24, 30, 6, 0) for the tag `NNP`;
- (17, 30, 7, 0) for the named entity `LOC`.

In SOLR, we store annotation tuples in the form of a *payload*, which is a chunk of data associated with the respective annotation posting. Tuples are stored in the uncompressed format and, therefore, occupy 16 bytes in the payload. In addition to this, each annotation is stored together with a compressed document ID and a compressed position of the annotation inside the annotation field. The latter positions are never used, but we have to store them, because SOLR payloads are enabled only for fields with positional information. In practice (see Table 4), however, this additional overhead is small compared to the size of uncompressed tuples.

```
POS_DT|0,1,1,0 POS_NN|2,9,2,0 POS_IN|10,12,3,0 POS_DT|13,16,4,0
POS_NNP|17,23,5,0
POS_NNP|24,30,6,0 NE_LOC|17,30,7,0
```

Table 2: A sample text that can be used to index POS tags and named entities for the phrase “A capital of the United States". The corresponding annotations are shown in Table 1.

To index documents in a SOLR index, one can submit documents in a special XML format via HTTP. In this format, we specify a text value for every document field. Payload data can be passed to SOLR by modifying text values for fields with configured payloads. Configuration consists in specifying a payload delimiter in the SOLR schema file as well as the name of the payload encoder class (inherited from `AbstractEncoder`). The payload encoder reads a textual representation of payload data and converts it to a binary form.

More specifically, one needs to append a payload delimiter followed by a payload string to every term in the field. Let the payload delimiter be the pipe-symbol and assume that the payload encoder uses the format in which an annotation tuple is specified via four comma-separated numbers. These numbers represent the start offset, the end offset, the annotation ID, and the parent annotation ID, respectively.

Then, to index payload data from Table 1, one should generate the text value shown in Table 2. Note that in this example the label of each field is prepended with a prefix: `POS_` for POS tags and `NE_` for the named entity. This is necessary to ensure that annotation terms belonging to different types are always different.

3.2 Retrieval Model

SOLR provides several entry points that can be used to extend functionality. In particular, it is possible to reimplement a search handler and a query parser, which is created by a custom query plugin. Overriding the search handler provides the most flexibility, because it permits one to implement an arbitrarily complex retrieval mechanism.

A simpler approach–chosen for this work–is to reimplement a query parser. Beside simplicity of implementation, it permits including our custom structured queries as a part of a standard SOLR query. In that, the results obtained from different parsers can be combined using Boolean operators `AND` and `OR`.

In this approach, each annotation subgraph is treated as a *meta-term*. The retrieval plugin computes the number of meta-term matches and passes this information to a SOLR similarity function. The latter aggregates global statistics of terms and annotations–e.g., by summing up inverted document frequency (IDF) values–and combines it with the number of in-document occurrences to compute the score of a document.

The main limitation imposed by the SOLR API is that the query plugin can only return a list of document IDs accompanied by document scores. Thus, it is not possible to return scored text extents in a way it is done by, e.g., Indri (without reimplementing the complete search handler).

```
1  _query_:"{!annographix ver=3
2              boost=3.2 span=1024 max_iter=200000
3             text_field=Text4Annotation annot_field=Annotation}
4              @srlp2:SRL_V  ~0:scored                    #covers(srlp2,0)
5              @srlp3:SRL_A1 ~1:100 ~2:points        #covers(srlp3,1,2)
6              @srlp4:SRL_A0 ~3:Wilt ~4:Chamberlain #covers(srlp4,3,4)
7              #parent(srlp2,srlp3,srlp4)"
```

Table 3: A sample structured query for the question “What year did Wilt Chamberlain score 100 points?" (TREC QA 2002 topic 1402). Line numbers are given only for presentation purposes.

3.3 Query Language for Structured Retrieval

A structured query can be included into another, e.g., standard, SOLR query. To this end, the structured query should have the following format:

```
_query_:"{!<plugin name> <plugin paramters>}
          <structured query> "
```

In words, it should have a prefix `_query_:"` followed by (1) query parameters surrounded by curly brackets and (2) by a query text concluded with the double quote.

Table 3 shows a sample structured query created for the question “What year did Wilt Chamberlain score 100 points?". It is designed to find all sentences with the verb `score` whose first argument (an agent) contains the words `Wilt` and `Chamberlain` while the second argument (patient or theme) contains the words `100` and `points`.

Line one in Table 3 provides the name of the plugin (starting with the exclamation mark) and an optional plugin version. The second line specifies the maximum length of a covering span (1024 *characters*), a boost value (an optional multiplier for document scores), and the maximum number of brute-force iterations that we carry out before giving up on checking constraints for one span. Instead of the explicitly specifying the size for the covering span, one can define the size implicitly, namely, by specifying the label of the covering annotation (parameter `cover_annot`). The maximum number of iterations is a parameter/threshold for the early termination heuristic described in Section 3.4.

Line two in Table 3 provides the name of the text field and the name of the corresponding field that stores text field's annotations. Lines four through seven represent the structured query itself. It has elements of three types: terms, annotations, and operators starting with the hash symbol. Terms and annotations define the nodes of an annotation graphs, while operators define graph edges.

A term is prefixed by the tilde, which is followed by an optional identifier, the colon, and, finally, by the term itself. The annotation has a similar structure. Yet, it is prefixed by the at-sign (which is followed by an optional identifier, the colon, and the annotation label).

Optional text and annotation identifiers are used to reference these query elements in operators. More specifically, the operator `#parent` defines a parent for one or more query element. We here use a functional notation to denote the parent identifier, followed by identifiers of one or more child elements. Identifiers are comma-separated and no spaces are currently allowed.

The operator `#covers` enforces the containment relationship and has the same syntax as the `#parent` operator. It tells the query plugin that the element represented by the first argument should cover elements represented by the second, third, etc. arguments.

For example, line 4 in Table 3 tells the query plugin that the term `scored` is contained within an annotation labeled `SRL_V`. Similarly, line 5 specifies that terms `100` and `points` are covered by the annotation `SRL_A1`; line 6 specifies that the annotation `SRL_A0` covers terms `Wilt` and `Chamberlain`; and line 7 requires annotations `SRL_A0` and `SRL_A1` to be children of the annotation `SRL_V`.

3.4 Algorithmic Details

As mentioned in Section 3.2, we opted for reimplementing a custom plugin that creates an instance of a custom query parser. In this approach, the query parser is responsible for creating instances of a weighting class, which, in turn, creates instances of a scorer class. These classes extend abstract classes `Weight` and `Scorer` from the package `org.apache.lucene.search`.

The scorer class is main workhorse whose instances find matching documents and compute a number matching textual extents inside these documents. SOLR employs a document-at-a-time evaluation strategy and the scorer class implements several functions to support this evaluation mode.

Two key functions are:

- `advance(docId)` – This function finds the first document (with ID at least as large as the function argument) containing all query terms and annotations. It moves posting pointers to a position where these pointers can be used to read positional information and payload data.

- `computeFreq()` – given that posting pointers are positioned at a document containing all query terms and annotations, compute the number of spans inside the document that match the query annotation graph.

To find a document containing all query terms and annotations, we use a classic "leap-frogging" algorithm similar to the algorithm already employed by the exact SOLR phrase scorer. The postings are first sorted in the order of increasing processing cost, which is typically estimated as a posting size (or a number of unique documents containing the term). Then, we iterate over the list with the smallest cost and advance pointers of more expensive postings to match the document of the first, i.e., the cheapest posting.[5]

If the advancement operation is efficiently implemented, which is claimed to be the case for SOLR API, this procedure effectively utilizes posting lists size differential. In a real life

[5]One possible optimization is to advance the pointer for the cheapest posting towards the largest previously seen document rather than always moving it by one position. However this optimization did not result in substantial time savings in our software.

scenario, this leap-frogging approach is one of the most fastest intersection methods, which, among others, outperforms the classic mergesort-like intersection algorithm [1, 12].

After finding a document containing all query terms and annotations, we read term offsets and annotations payloads, store them in-memory arrays (note that annotation payloads are extracted using a custom payload decoder) and invoke the function `computeFreq`. The latter creates a span iterator to explore all segments of texts that may match the query annotation sub-graph.

The implementation of the span iterator is trivial, if the span is defined by a covering annotation. In this case, we merely loop over all annotation elements already stored in the corresponding array. To find all segments containing given terms in a window of limited size, we use a well-known plane-sweep algorithm [20] implemented using a priority queue.

The spans are explored in the order of non-decreasing start offsets. For each span, we align query terms and annotations against this span. More specifically, for every query element (i.e., a term or annotation) we find the first and the last index inside the corresponding memory array such that the start offset of the query element is inside the span. Alignment relies on an exponential search [2], also known as galloping.

For each term and annotation, these indices define a subset of query elements that are potentially fully covered by the span. We exhaustively consider all combinations of terms and annotations from these sets and checks that they (1) all fit into the span and (2) do not violate parent-child or containment constraints expressed by the query.

Because we do not need to compute the number of matches inside the span–but rather verify if a match exists or not–we finish checking the current span, and move to another span right after the first match is found. In that, we also increment a frequency counter.

To make this brute-force verification feasible, several heuristics are used. First, we order terms/annotations in the order of decreasing number of query elements connected to a given term/annotation through the annotation graph. Ties are resolved by placing less frequent elements before more frequent ones. To compute the number of connected elements, we consider the query graph to be undirected (both the `#cover` and `#parent` operators are seen as defining undirected edges).

Second, if the query graph is not fully connected, we evaluate each disjoint component separately. In that, components are checked in the order of decreasing size. If there is a tie, the component having the least frequent query element is checked first. The idea behind these heuristics is that, if the combination of query elements is incorrect, graph constraints are more likely to be violated for well-connected rare query elements.

These two heuristics were sufficient to answer simple queries–such as the query shown in Table 3–in the order of 10 millisecond for the AQUAINT collection. In particular, less than 50 milliseconds (on average) was needed to execute queries generated from answer-bearing sentences (see Section 4.3.2). Yet, the query plugin was way too slow on complicated queries specifying the (almost) complete annotation graph of a sentence.

For this reason, we introduced an early termination heuristics by putting a limit on the maximum number of brute-force iterations allowed in a single span. Whenever this threshold was reached, we terminated the check of the current span and proceeded to another one. This threshold was specified as the query plugin parameter `max_iter`. The effect of using this heuristic is analyzed in Section 4.3.3.

4 Experiments

4.1 Data Sets

We used the AQUAINT data set [10]. This data set contains 1,033,162 English news articles (roughly 375 million words; 3GB of data) from three news sources: Associated Press (1998–2000), New York Times (1998–2000), Xinhua News Agency (1996–2000).

To evaluate performance, we created two types of structured queries. Queries of the first type were used to evaluate effectiveness of structured retrieval queries for document retrieval in a QA system. The methodology was similar to that of Bilotti [3] in that these queries were created from answer-bearing sentences.

Answer-bearing sentences were first annotated using the semantic role labeler SENNA. Then, the annotation graph was pruned to contain only terms from the original questions (terms were considered identical if they had identical lemmas according to the Stanford CoreNLP lemmatizer). This pruning procedure will be referred to as a *projection* of answer-bearing sentences to original queries. The remaining sub-graph was converted into a query in a straightforward fashion, using operators that specify parent-child and containment relationship.

Similar to Bilotti [3], we attempted to enhance queries with answer type placeholders. To this end, we predicted the question answer-type using the question-analysis module of the Open Ephyra [21]. If the type of the answer was a person, an organization, or a location, we added to the query annotations `@:NE_PER`, `@:NE_ORG`, or `@:NE_LOC`. This instructed our plugin to return only those text segments that contained a named entity matching the expected answer type. This substantially (by more than 20%) degraded retrieval performance and we, therefore, did not use answer-type placeholders in the final evaluations.

Unfortunately, the answer-bearing sentences collected by Bilotti [3] were lost and, hence, we needed to recreate them. Note that the set of answer-bearing sentences was collected prior before experiments were carried out. Once created, it was never modified.

Similar to the approach of Bilotti, we retrieved potentially answer-bearing documents using document-level relevance judgements specifically created for document retrieval task (in a QA system). Then, we used patterns provided by TREC organizers to identify sentences containing an answer text (sentence boundaries were detected using Stanford CoreNLP).

This procedure generated a lot of false positives, because many of the sentences contained the answer text by chance (e.g., a name of the city), without actually answering the question. In addition, some sentences answered the question only partially and it was not possible to derive the answer without consulting additional sources or without making additional assumptions. We used Bilotti's guidelins [3] to cull out false positives as well as all insufficiently specific and/or supportive sentences. Additionally we removed a few duplicate sentences that appeared in more than one document.

Document-level relevance judgments were taken from two sources: the "MIT 109" test set and TREC QA 2005 document ranking task containing 109 and 50 factoid questions, respectively. In the case of "MIT 109", document-level relevance judgments were created by substantially expanding the set of relevance judgements of TREC 2002 QA task [14]. Because of this expansion, the average number of known relevant documents per question increased from 6 to 18. In the case of TREC QA 2005 task, there are 50 questions with about 31 relevant document per question on average.

However, each question in TREC QA 2005 task is a part of a longer series of questions, where questions may use a pronoun (or other referential expressions) to include answers or noun phrases from the preceding questions and/or a series description. Systems participating in TREC QA 2005 document retrieval task were supposed to resolve all references automatically. We, however, simplified the task and resolved references manually. For example, the original question 71.4 “Where is this company based?" references the company name that is an answer to question 71.3 “Who manufactures the F16?". Hence, question 71.4 was manually replaced with “Where is Lockheed Martin based?".

Overall we could find 1164 answer-bearing sentences: 630 for the "MIT 109" data set and 534 for the TREC QA 2005 document ranking set (For the "MIT 109" data set Bilotti [3] was able to find 619 answer-bearing sentences). These sentences corresponded to 121 question, i.e., it was not possible to find answer-bearing sentence for all questions. However, we used only 116 questions, leaving out five questions where our query reformulation procedure (see Section 4.2) did not generate a reformulation.

Queries of the second type were used to evaluate retrieval efficiency as well as the loss of recall due to the early termination heuristic. They were created using 1000 randomly sampled sentences from the AQUAINT collection. For each of the sentence, we generated four queries that represented the complete annotation graph with respect to one or more annotation type (stop words were not included). Because the query language allows us to explicitly specify both the graph nodes and edges, mapping of graphs to queries is a straightforward operation.

Three queries were created for POS tags, semantic role labels (SRLs), and dependency relations, respectively. The fourth query included annotations of all three types. POS tags do not have parent-child links and, therefore, they represented simpler queries. Queries that included all three annotation types were the hardest.

4.2 Setup

All experiments were carried out on a quad-core core-i7 laptop with 16 GB of DDR3 memory, which ran OS Linux.

Performance of retrieval methods was evaluated using the mean average precision (MAP). Similar results were obtained when performance was measured using the mean reciprocal rank. The version of Apache SOLR was 4.6. The similarity model was BM25 [19].

The AQUAINT collection was annotated using the SENNA parser [6] to obtain POS tags, named entities, SRLs, and syntax trees. Syntax trees were fed to the Stanford CoreNLP toolkit [15] to produce dependency tress (syntax trees were subsequently discarded). Additionally the Stanford CoreNLP was employed for for sentence detection, lemmatization, and POS tagging (POS tagging was done for the query analysis).

NLP pipelines were implemented on top of the UIMA ECD[6], which is an extension of the UIMA framework.[7] All the major code for this project (pipelines, indexing software, and the SOLR plugin) was written in the Java language. The SOLR Annographix plugin is

[6]`https://github.com/oaqa/uima-ecd`

[7]`https://uima.apache.org/`

Dict.	Freq. + skips	Positions	Offsets	Annot. tuples	**Total**
0.07	0.44	1.91	0.7	**19.7**	**22.8**

Table 4: Index Size Statistics (in GBs)

publicly available online.[8]

We used a query reformulation module (originally implemented for another project). For example, given interrogative queries in the form “Where is Eiffel Tower?” and an answer “Paris”, the query reformulation module generated a declarative statement “Eiffel Tower is located in Paris". The use of this module consistently and substantially (by about 10%) improved performance of *all* unstructured and structured retrieval models.

The query reformulation module can be seen as a Java reimplementation of the Lin’s [13] query reformulation module developed for the QA system Aranea (written in Perl). Similar to Aranea, it relied on POS tagging, although we did not use several inexact heuristics that tend to generate mostly ungrammatical reformulations. This simplification came at a price: we could not generate reformulations for about 10% of TREC QA questions.

4.3 Results

4.3.1 Index Size

In addition to the text of AQUAINT news articles, we indexed POS tags, named entities, SRLs, dependency trees, as well as annotations delineating sentence and paragraph boundaries. It took one hour to index approximately 0.375 billion regular terms and 1.321 billion annotations in an single-thread mode.

In total, the index occupied almost 23 GBs (see Table 4). Most of the space was consumed by uncompressed annotation tuples (each of which was 16 bytes).

Somewhat unexpectedly, term offsets used much less space than term positions. The reason is that offsets were stored only for the text fields, but positions needed to be stored for both the text and the annotation field. Because there were three times as many annotations as text terms, the overall number of positions to be indexed was 4x the number of offsets.

4.3.2 Answer-Bearing Sentence Queries

In this section, we focus on the task of document retrieval using unstructured and structured queries. The structured queries were created from answer-bearing sentences as described in Section 4.1.

Similar to Bilotti [3], we considered a *single structure* and an *every structure* scenario. The difference is that Bilotti concentrated on a sentence retrieval task, while we evaluated the effectiveness of document retrieval. In fact, as explained in Section 3.2, we cannot support retrieval of a sub-document textual segments in SOLR by reimplementing only the query parser/plugin.

[8]`https://github.com/oaqa/annographix`

```
1  _query_: "{!edismax mm=3 ps=20} wilt chamberlain score 100 point
   " OR
2   "Wilt Chamberlain"~10^0.3 OR "Wilt score"~10^0.3        OR
3   "Chamberlain score"~10^0.3 OR "Chamberlain 100"~10^0.3 OR
4   "score 100"~10^0.3           OR  "score points"~10^0.3     OR
   "100 points"~10^0.3
```

Table 5: The query generated for the query reformulation "Wilt Chamberlain scored 100 points". Line numbers are given only for presentation purposes.

In the single structure scenario, each answer-bearing sentence and a corresponding query are considered to represent a unique topic, for which there is exactly one relevant document. This document contains the answer-bearing sentence.

In the every structure scenario, queries created from answer-bearing sentences are grouped by the original TREC QA question. Therefore, for each of the 116 TREC QA questions used in our evaluation (see Section 4.1 for query selection details), we usually have multiple queries. Each of this queries is executed and results are combined.

To combine results, we merged document IDs returned by several queries and sorted them by their retrieval scores. If the same document ID was returned by more than one query, we used the maximum of the scores. We also tried the round-robin combination approach, but it always delivered inferior performance (about 10% worse than the simple max).

We experimented with several types of unstructured queries and found that:

- A simple bag-of-words SOLR query (without any proximity operators) is outperformed by a *sloppy-phrase* query. The sloppy phrase retrieval model favors, but not enforces, occurrences of query terms within a text window of a given size;
- An unstructured query that is based on the result of query reformulation is more effective than either the original question or the query obtained by the projection of an answer-bearing sentence to the corresponding query;
- Performance of sloppy-phrase queries can be further improved by OR-ing them with phrasal queries constructed using original query bigrams. This approach is somewhat similar to the term-dependency model [16].

A sample query generated for the query reformulation "Wilt Chamberlain scored 100 points" is presented in Table 5. The first line shows a sloppy-phrase sub-query that returns documents containing at least three terms (parameter `mm=3`) out of the specified four. It favors occurrences where terms are found in the window of the size 20 (parameter `ps=20`). It is combined (via the `OR` operator) with several sub-queries that search for ordered query bigrams within the window of the size 10. The score of any bigram occurrence is multiplied by 0.3.

	Every structure	Single structure
	MAP	MAP
baseline	0.25	0.07
combined	0.33 (+32%)	0.13 (+86%)

Table 6: Comparison of MAP for Single and Every Structure scenarios

To sum up, baseline non-structured queries were created by carrying out query reformulation, creating a sloppy-phrase query, and enhancing it by bigram queries as shown in Table 5. There are several parameters (such the bigram boost value or the size of the window) that were manually tuned for the complete test set of questions, to maximize the MAP. Optimizing baseline performance on the given test set makes it harder for other methods to outperform the baseline.

To verify if unstructured queries can be improved by structured retrieval capabilities, we OR-ed unstructured queries with structured queries generated from answer-bearing sentences. The obtained queries had two parameters: the boost value for a structured query and the length of the span (see Table 3). The optimal parameter values were obtained by an eight-fold *cross validation*.

The results are presented in Table 6. Performance was measured using the MAP; both p-values were smaller than 0.001 (computed using the permutation test [24]). It can be seen that queries with linguistically-motivated constraints outperformed unstructured queries. In that, the differences were substantial and statistically significant.

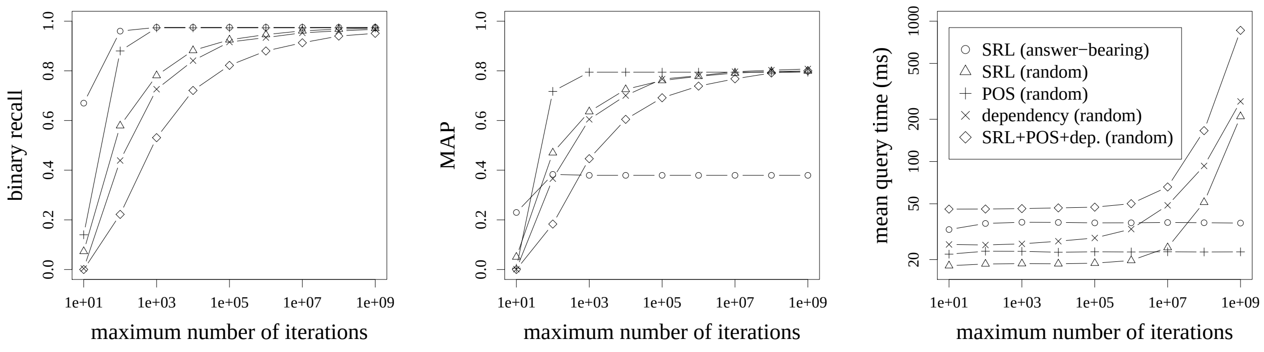
We conclude this subsection by noting that, because we evaluated document rather than sentence retrieval and used a different subset of queries, the results are not directly comparable to results in Table 5.1 of Bilotti [3].

4.3.3 Efficiency-Effectiveness Tradeoffs

To study how the choice of the early termination threshold affects retrieval speed and accuracy, we experimented with both types of queries. More specifically, we used all the SRL queries generated from (200) answer-bearing sentences as well as 1000 queries of the second type. Recall, that the latter represented annotation graphs of randomly selected corpus sentences (see Section 4.1). Queries of the second type are often quite complex. For example, if the query represents a complete graph of dependency relations, it has one containment operator and one parent-child relationship operator for each non-stop word.

For each set of queries we varied the maximum number of iterations within a span, i.e., the early termination threshold, and measured three parameters: the MAP, the binary recall, and the mean retrieval time for the in-memory index scenario. Effectiveness was measured as in the *single structure* experiment (see Section 4.3.2). Specifically, each sentence was considered to be a unique topic with a single relevant document (containing this sentence).

To simulate retrieval for the case when the index is fully loaded into memory, we ran each query twice and recorded time only for the second run. In that, we also ensured that the SOLR result cache was fully disabled and that the query plugin processed the query every



(a) Binary recall (b) MAP (c) Retrieval Time (ms)

Figure 1: Efficiency and effectiveness plots for various values of the early termination threshold.

time we sent a request to the SOLR server.[9] The results are given in Figure 1.

First, consider queries generated from answer-bearing sentences, which are labeled as as *SRL (answer-bearing)*. These sentences are simple and do not require more than one thousand brute-force iterations for the verification of a single occurrence. However, the MAP is rather low (see Figure 1b), because these queries are less specific than queries generated from complete annotation graphs (without projecting to the TREC QA topics).

Queries of the second type have the MAP as high as 0.8, but they can be more expensive to evaluate. For example, effectiveness of queries generated from dependency graphs grows substantially till the maximum number of iterations reaches 10^6. Beyond this value, we observe only marginal improvements in effectiveness, while the retrieval time increases substantially.

To sum up, for all types of queries, the threshold value of 10^7 provides the binary recall at least as high as 0.9. In that, the mean retrieval time is less than 100 ms. Thus, our implementation is not ideal, but it can be useful even in its current form.

5 Conclusions

We implemented SOLR Annographix, which is a software suite for indexing and querying annotation graphs generated by NLP pipelines. The effectiveness and efficiency of the software was evaluated in the retrieval tasks where queries were created:

- by projecting annotation graphs of answer-bearing sentences to TREC QA topics;
- from annotation graphs of randomly chosen sentences without projection.

[9]To be 100% sure, we logged all events when our query parser was instantiated and verified that the log entries appeared in the log when a query was repeated.

In the document retrieval task, SOLR Annographix outperformed a proximity-enhanced bag-of-words model. In this scenario, queries were simple and could have been evaluated efficiently without the early termination heuristic.

In the task of retrieving sentences using their unprunned annotation graphs, queries were more complex and the early termination were crucial to efficient processing. Yet, it was possible to achieve the binary recall of 0.9 while answering queries in less than 100 ms on average.

We believe that the current software can already be useful. Yet, it will benefit from the following enhancements:

- The simplistic constraint verification algorithm should be replaced by an algorithm with better worst-case performance guarantees, which, ideally, would not require an early-termination heuristic. There is a number of ways to achieve this goal, yet, a corresponding discussion is beyond the scope of this report.
- Annotations can outnumber original text terms. Thus, the current scheme where each annotation tuple uses 16 bytes is quite wasteful. The space consumption can be reduced by using, e.g., a simple frame-of-reference scheme [11].
- Our query language is expressive and allows one to directly specify the annotation sub-graph in terms of nodes and connecting edges. Yet, it is not always convenient for humans to use. Thus, this issue needs to be addressed.

References

[1] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *Journal of Experimental Algorithmics (JEA)*, 14:7, 2009.

[2] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.

[3] Matthew W Bilotti. *Linguistic and semantic passage retrieval strategies for question answering.* PhD thesis, Carnegie Mellon University, 2009.

[4] Matthew W Bilotti, Paul Ogilvie, Jamie Callan, and Eric Nyberg. Structured retrieval for question answering. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 351–358. ACM, 2007.

[5] Jennifer Chu-Carroll, James Fan, BK Boguraev, David Carmel, Dafna Sheinwald, and Chris Welty. Finding needles in the haystack: Search and candidate generation. *IBM Journal of Research and Development*, 56(3.4):6–1, 2012.

[6] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[7] Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question answering passage retrieval using dependency relations. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 400–407. ACM, 2005.

[8] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10:33–58, 2012.

[9] Gang Gou and Rada Chirkova. Efficiently querying large xml data repositories: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1381–1403, 2007.

[10] David Graff. The AQUAINT Corpus of English News Text LDC2002T31. Philadelphia: Linguistic Data Consortium, 2002.

[11] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[12] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *arXiv preprint arXiv:1401.6399*, 2014.

[13] Jimmy Lin. An exploration of the principles underlying redundancy-based factoid question answering. *ACM Transactions on Information Systems (TOIS)*, 25(2):6, 2007.

[14] Jimmy Lin and Boris Katz. Building a reusable test collection for question answering. *Journal of the American Society for Information Science and Technology*, 57(7):851–861, 2006.

[15] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[16] Donald Metzler and W Bruce Croft. A markov random field model for term dependencies. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 472–479. ACM, 2005.

[17] Yusuke Miyao, Tomoko Ohta, Katsuya Masuda, Yoshimasa Tsuruoka, Kazuhiro Yoshida, Takashi Ninomiya, and Jun'ichi Tsujii. Semantic retrieval for the accurate identification of relational concepts in massive textbases. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 1017–1024. Association for Computational Linguistics, 2006.

[18] John M Prager. Open-domain question-answering. *Foundations and Trends in Information Retrieval*, 1(2):91–231, 2006.

[19] Stephen Robertson. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of documentation*, 60(5):503–520, 2004.

[20] Kunihiko Sadakane and Hiroshi Imai. Text retrieval by using k-word proximity search. In *Database Applications in Non-Traditional Environments, 1999.(DANTE'99) Proceedings. 1999 International Symposium on*, pages 183–188. IEEE, 1999.

[21] Nico Schlaefer, Jeongwoo Ko, Justin Betteridge, Manas A Pathak, Eric Nyberg, and Guido Sautter. Semantic extensions of the ephyra qa system for trec 2007. In *TREC*, 2007.

[22] Dan Shen and Mirella Lapata. Using semantic roles to improve question answering. In *EMNLP-CoNLL*, pages 12–21. Citeseer, 2007.

[23] Robert F Simmons, Sheldon Klein, and Keren McConlogue. Indexing and dependency logic for answering english questions. *American Documentation*, 15(3):196–204, 1964.

[24] Mark D Smucker, James Allan, and Ben Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 623–632. ACM, 2007.

[25] Hugo Zaragoza, Michael Matthews, Roi Blanco, and Jordi Atserias. Annotated search and element retrieval. ISWC, 2009.