Development of a Search Engine and Applications in IR

Robert Pinsler
WKWSCI, NTU
N1509281G@e.ntu.edu.sg

Yiding Liu
SCE, NTU
LIUY0130@e.ntu.edu.sg

Yitong Guan
SCE, NTU
GUAN0049@e.ntu.edu.sg

Jenn Bing Ong
SCE, NTU
ONGJ0063@e.ntu.edu.sg

ABSTRACT

1. INTRODUCTION

2. SYSTEM OVERVIEW

The search engine is able to index and search publication records listed in the dblp computer science bibliography (DBLP) [4]. It is built on top of PyLucene, a Python extension of the open-source software library Lucene [1] that provides text indexing and search capabilities.

Publication records from DBLP are provided in a single XML file. The dataset comprises of various kinds of record types, such as articles published in a journal or magazine (*article*), papers published in a conference or workshop proceedings (*inproceedings*), books, and PhD theses. From those, only documents that are classified as *article* or *inproceedings* are considered.

3. INDEXING

To index a record, it first has to be extracted from the XML file before it can be stored. This can be done conveniently using an XML parser. Due to the large file size of the dataset, reading in the whole XML tree at once is impractical. Instead, we use a SAX-like parser from the lxml Python library [2] that sequentially reads the document and emits events when it encounters certain XML tags. This allows to only react to <*article*> and <*inproceedings*> tags, thereby ignoring record types we are not interested in. This makes it very fast and memory-efficient. During event processing, it is possible to access the current element, i.e. an *article* or *inproceedings* record, and its children, i.e. attributes of the record such as title or year of publication. As some of the publication titles use basic HTML text formatting, HTML tags are removed prior to indexing.

Each record is indexed as a Lucene *document*, comprising of the following *fields*: id, title, authors, year and venue (journal or booktitle, depending on the record type). To facilitate the search, an additional *content* field is added that concatenates the values of the above fields except the id. All fields are treated as strings[1]. Contents of the original fields (i.e. without the *content* field) are stored in the index, which allows to retrieve them later during search. This is especially useful to enhance search results with additional information when presented to the user. The values of the fields *id* and *year* are stored as-is. All other fields are tokenized (f_{token}) and further processed, including lowercase conversion (f_{low}), removal of stopwords (f_{stop}) and stemming (f_{stem}). In particular, we use the Porter stemmer and the default stopword list of Lucene, which consists of 33 common English words. Table 1 summarizes the analysis applied to each field.

In order to evaluate the impact of additional token processing onto the index, different configurations of lowercase conversion, removal of stopwords and/or stemming are applied to the *title* field. We measure the speed of indexing in seconds, t_{ind}, as well as the number of terms in the vocabulary, $|V_{\text{title}}|$. The results are shown in Table 2. [...]

Table 1: Analysis applied to document fields

Field	f_{token}	f_{low}	f_{stop}	f_{stem}
id				
year				
authors	X	X	X	
venue	X	X	X	
content	X	X	X	
title	X	X	X	X

Table 2: Evaluation of token processing on *title* field

f_{low}	f_{stop}	f_{stem}	t_{ind}	$\|V_{\text{title}}\|$
			0.123	1337
X			0.123	1337
	X		0.123	1337
		X	0.123	1337
X	X	X	0.123	1337

4. SEARCH

The search module makes use of the document index to retrieve the most relevant documents to a given query. It supports free text keyword queries on any combination of the attributes title, authors, year and venue. By default, all

[1]The *year* attribute may also be stored as integer, allowing for features like range searches. However, this would require additional effort during search that we try to avoid.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

attributes are considered. This is achieved by utilizing the *content* field of the index. We refer to this as a standard search. Additionally, it is possible to query a combination of specific fields through an advanced search. Those keywords are directly matched against the respective fields in the index. In this case, documents must contain the provided keywords to be returned in the result set. Standard and advanced search can be freely combined during a single search request.

Phrase queries are supported using double quotation marks (e.g. “information retrieval”) in both standard and advanced search. Documents containing a particular phrase receive a higher weight during scoring. This requires an exact match up to the processing on the respective field. If a document does not contain the phrase, it may still be considered when there is a match for at least one of the keywords within the phrase. This considerably increases recall while risking an increase of false positives. Note that there is no restriction for the number of phrases within a query.

The search returns the N most relevant documents along with their ranks, scores, ids and snippets, where N is a configurable parameter. Relevance is determined based on Lucene’s internal scoring function. It also measures the time needed to retrieve the results. The complete procedure for handling queries is outlined in Listing 1.

Listing 1: Query handling

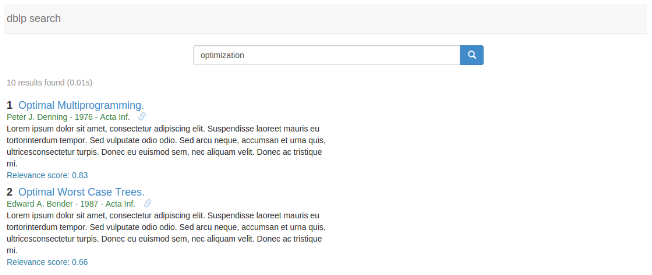
```
def search(self, query, adv_query=None, N=0):
  query = query.strip()
  bq = BooleanQuery()
  if query != '':
    # handle phrases in standard search
    pq, query = self.get_pq(query, 'content')
    if pq is not None:
        bq.add(pq, BooleanClause.Occur.MUST)
  if query != '':
    # handle remaining keywords
    qparser = QueryParser(self.VERSION,
                          'content',
                          self.analyzer)
    q = qparser.parse(query)
    bq.add(q, BooleanClause.Occur.SHOULD)
  if adv_query is not None:
    for field, query in adv_query.iteritems():
      # handle phrases in advanced search
      pq, query = self.get_pq(query, field)
      if pq is not None:
        bq.add(pq, BooleanClause.Occur.MUST)
      if query != '':
        # handle remaining keywords
        qparser = QueryParser(self.VERSION,
                              field,
                              self.analyzer)
        q = qparser.parse(query)
        bq.add(q, BooleanClause.Occur.MUST)
  start = time.clock()
  docs = self.searcher.search(bq, N).scoreDocs
  end = time.clock()
  duration = end-start
  return docs, duration

def get_pq(self, q, field, slop=0, boost=5):
  phrases = re.findall(r'"([^"]*)"', q)
  if len(phrases) == 0:
    return None, q

  q = re.sub(r'"([^"]*)"', "", q).strip()
  bq = BooleanQuery()
  for phrase in phrases:
    qparser = QueryParser(self.VERSION,
                          field,
                          self.analyzer)
    # handle phrase and single keywords
    pq = qparser.parse('%s "%s"~%d^%.1f' %
                       (phrase, phrase,
                        slop, boost))
    bq.add(pq, BooleanClause.Occur.MUST)
  return bq, q
```

5. EVALUATION

6. USER INTERFACE

There are two ways to interact with the search engine: via a text-based command line interface or via a web UI. The latter is built with Flask [5], a leightweight Python web framework that leverages Jinja2 [6] as a templating engine. This allows to easily create HTML documents from within Python. Additionally, we incorporate the web framework Bootstrap [3] to achieve responsive web design. Figure 1 depicts the search results for an example query.

Figure 1: Web-based search user interface



10 results found (0.01s)

1 Optimal Multiprogramming.
Peter J. Denning - 1976 - Acta Inf.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse laoreet mauris eu tortorinterdum tempor. Sed vulputate odio odio. Sed arcu neque, accumsan et urna quis, ultricesconsectetur turpis. Donec eu euismod sem, nec aliquam velit. Donec ac tristique mi.
Relevance score: 0.83

2 Optimal Worst Case Trees.
Edward A. Bender - 1987 - Acta Inf.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse laoreet mauris eu tortorinterdum tempor. Sed vulputate odio odio. Sed arcu neque, accumsan et urna quis, ultricesconsectetur turpis. Donec eu euismod sem, nec aliquam velit. Donec ac tristique mi.
Relevance score: 0.66

7. APPLICATIONS IN IR

7.1 Popular research topics

7.2 Similar publication venues

8. CONCLUSIONS

9. REFERENCES

[1] Apache Software Foundation. Lucene. https://lucene.apache.org, 2016. last accessed: 18-March-2016.

[2] S. Behnel et al. lxml. http://lxml.de, 2016. last accessed: 18-March-2016.

[3] Bootstrap Core Team. Bootstrap. http://getbootstrap.com, 2016. last accessed: 18-March-2016.

[4] M. Ley et al. dblp computer science bibliography dataset. http://dblp.uni-trier.de/xml, 2016. last accessed: 18-March-2016.

[5] A. Ronacher. Flask. http://flask.pocoo.org, 2016. last accessed: 18-March-2016.

[6] A. Ronacher. Jinja2. http://jinja.pocoo.org, 2016. last accessed: 18-March-2016.