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

This study reports the development of a search engine that consists of indexing the dblp computer science bibliography data and process queries on the data fields in terms of keyword and phrase search. The relevance of the query results is evaluated and a user interface for the search engine is built. The most popular research topic and most similar publication venue and year are retrieved as part of the applications in information retrieval.

CCS Concepts

•**Information systems** → **Document filtering;** *Information extraction;* Relevance assessment;

Keywords

Search engine; information retrieval; Lucene; DBLP; indexing; query processing; user interface

1. INTRODUCTION

A search engine is an Information Retrieval (IR) system designed to help users find information stored on a computer system. In particular, a web-search engine is designed to search for information on the World Wide Web. The web was initially indexed by human until the 1990s when more and more web servers went online. Human-maintained lists are subjective and expensive to build and maintain, thus cannot scale the web effectively. It was not until a landmark paper by Google [**?**] that addresses both the issues of search quality and scalability of an automated web-search engine. Google introduced PageRank, uses the anchor text, proximity information, etc to improve the search quality dramatically and built a practical and automated web-search engine by fast crawling and query processing technology as well as robust database and operating system [**?**].

The web is a vast collection of completely uncontrolled heterogeneous documents; web documents have extreme variation both in terms of internal and external meta information. Big data refers to datasets that limits the capacity of traditional data processing systems to conduct effective analysis in terms of its data volume, quality, variety in data representation / structure, and acquisition speed; hence requiring new processing technologies [**?**]. All these add to the complexity of IR, which pose enormous challenge but presents unprecedented opportunity for new knowledge discovery in real-time that improves decision making.

The objective of this study is to learn and practice the building blocks of a basic search engine using open-source Application Programming Interface (API) and applications in Information Retrieval (IR). The organization of this report is as follows. Recent development of search engine and IR are covered in Section 2. Section 3 provide information of the experimental setup such as dataset and API used. The results of the search engine built and IR applications are given in Section 4 with conclusion in Section 5.

2. RECENT DEVELOPMENT

Elaborate on the recent development in search engine and information retrieval: personalised mobile search engine [...]

3. EXPERIMENTAL SETUP

Elaborate on DBLP dataset, Lucene API

4. SYSTEM OVERVIEW

The search engine is able to index and search publication records listed in the dblp computer science bibliography (DBLP) [2]. 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.

5. INDEXING

Indexing the DBLP dataset is a multi-step process. First, each record has to be extracted from the XML file before it can be further processed. Additional preprocessing is applied to match the expected input format. Finally, the

[1]https://lucene.apache.org

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

records are processed and indexed through the PyLucene API. Listing 1 gives an overview of the Indexer class that provides above functionality. It takes as input the location of the file to be indexed, the desired index storage location and an analyzer instance that we will introduce later. By calling its own index function, it triggers the indexing process (Listing 1, line 12). In the following, we explain each step in more detail.

```
1  TAGS = ('article', 'inproceedings')
2  VERSION = Version.LUCENE_CURRENT
3  CREATE = IndexWriterConfig.OpenMode.CREATE
4
5  class Indexer():
6    def __init__(self,data_dir,store_dir,analyzer):
7      store = SimpleFSDirectory(File(store_dir))
8      config = IndexWriterConfig(VERSION,analyzer)
9      config.setOpenMode(CREATE)
10     self.writer = IndexWriter(store,config)
11     self.htmlparser = HTMLParser()
12     self.index(data_dir)
13     self.writer.close()
14
15   def index(self,data_dir):
16     context = etree.iterparse(data_dir,tag=TAGS,
17       events=('end',),dtd_validation=True)
18     for event,elem in context:
19       self.index_document(elem)
20
21     def index_document(self,elem):
22       # indexes extracted element
```

Listing 1: Indexer class[2]

Parsing We use an XML parser to extract the DBLP records from the dataset. Due to the large file size, reading in the whole XML tree at once is impractical. Instead, we use a SAX-like parser from the lxml[3] Python library 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. For each emitted event, we call the index_document function. (Listing 1, lines 15-19)

```
1  FIELDS =
       ['title','author','year','journal','booktitle']
2  HTML_TAGS = ('i','ref','sub','sup','tt')
3
4  def index_document(self,elem):
5    etree.strip_tags(elem,HTML_TAGS)
6    doc = Document()
7    f = StringField('id',elem['key'],Store.YES)
8    doc.add(f)
9    for ch in elem:
10     if ch.tag not in FIELDS:
11       continue
12     if ch.text is None:
13       ch = etree.tostring(ch)
14       ch = self.htmlparser.unescape(ch)
15       ch = etree.fromstring(ch)
16
17     if ch.tag == 'title':
18       f = TextField('title',ch.text,Store.YES)
19     elif ch.tag == 'author':
20       f = TextField('authors',ch.text,Store.YES)
21     elif ch.tag == 'year':
22       f = StringField('year',ch.text,Store.YES)
23     else:
24       f = TextField('venue',ch.text,Store.YES)
25
26     doc.add(f)
27     cf = TextField('content',ch.text,Store.NO)
28     doc.add(cf)
29   self.writer.addDocument(doc)
```

Listing 2: index_document function

Preprocessing Given the current element, i.e. an *article* or *inproceedings* record, we can now iterate through its children, i.e. record attributes like title or publication year. This is shown in Listing 2. As some records contain HTML formatting, irrelevant tags are removed and HTML-encoded characters are replaced (Listing 2, lines 5 and 12-15).

Indexing Each record is indexed as a Lucene *document*, comprising of the following *fields*: id, title, authors, year and venue. The venue field is derived from the journal or booktitle attribute, 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. This can be easily achieved by subsequently adding all values to the same field. This method is also used to index multiple authors. (Listing 2, lines 19-20 and 27-28).

All fields are treated as strings[4]. 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, which is a property of the StringField class. All other fields are tokenized (f_{token}) and further processed.

The processing applied to each of those TextFields is determined by the analyzer that is used, which is known by the index writer (Listing 1, lines 6-10 and Listing 2, line 29). By default, all TextFields are processed in the same way, which may include 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. In order to assign different analyzers depending on the field, we utilize PyLucene's PerFieldAnalyzerWrapper as shown in Listing 3. In this case, we apply the StandardAnalyzer from PyLucene to perform lowercase conversion and stopword removal on each tokenized field except the title, where the CustomAnalyzer introduced in Listing 4 is used.

```
1  INDEX_DIR = 'index'
2  DATA_DIR = 'data/dblp.xml'
3
4  config = {'lowercase':True,'stemming':True,
       'stopwords':True}
5  title_analyzer = CustomAnalyzer(config)
6  per_field = HashMap()
7  per_field.put("title",title_analyzer)
8  default = StandardAnalyzer(VERSION)
9  analyzer =
       PerFieldAnalyzerWrapper(default,per_field)
10 Indexer(DATA_DIR,INDEX_DIR,analyzer)
```

Listing 3: Indexing with field-dependent analyzers

Evaluation 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. For this, we make use of our CustomAnalyzer that gives us full control over those configurations by extending the base analyzer of PyLucene.

[2]Listings are slightly modified to improve readability.
[3]http://lxml.de
[4]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.

The class could be easily extended to support alternative stemmers and stopword lists by further parametrizing the config parameter.

```
class CustomAnalyzer(PythonAnalyzer):
  def __init__(self,config):
    self.lowercase = config['lowercase']
    self.stemming = config['stemming']
    self.stopwords = config['stopwords']
    PythonAnalyzer.__init__(self)

  def createComponents(self,fieldName,reader):
    src = StandardTokenizer(VERSION,reader)
    fltr = StandardFilter(VERSION,src)
    if self.lowercase:
      fltr = LowerCaseFilter(VERSION,fltr)
    if self.stemming:
      fltr = PorterStemFilter(fltr)
    if self.stopwords:
      sw = StopAnalyzer.ENGLISH_STOP_WORDS_SET
      fltr = StopFilter(VERSION,fltr,sw)
    return self.TokenStreamComponents(src,fltr)
```

Listing 4: CustomAnalyzer class

During our experiments, we measure the speed of indexing (including parsing and analyzing the documents) in seconds, t_{ind}, as well as the number of terms in the vocabulary, $|V_{\text{title}}|$. The results are shown in Table 1. Note that indexing speed slightly varied over different runs (not shown).

Table 1: Evaluation of token processing on *title* field

f_{low}	f_{stop}	f_{stem}	t_{ind}	$\lvert V_{\text{title}}\rvert$
			417.97 (100%)	436 354 (100%)
X			430.57 (+3%)	346 286 (−21%)
	X		414.89 (−1%)	436 321 (±0%)
		X	439.75 (+5%)	365 534 (−16%)
X	X	X	417.94 (±0%)	288 710 (−34%)

Applying lowercase conversion or stemming reduces the vocabulary significantly, whereas the removal of stopwords has nearly no impact. This is expected, since the first two techniques are able to map multiple tokens onto the same term whereas the exclusion of stopwords decreases the vocabulary size only by the number of stopwords. By combining multiple processing steps, the vocabulary can be further reduced, leading to a reduction of to up a third of the original size when no further processing is applied. In terms of indexing speed, we found that the additional processing steps are negligible. In particular, there seems to be no positive correlation between indexing speed and the number of processing steps that are applied. One reason for this is that while processing each document is more time-consuming, the cost for indexing a processed document is actually reduced. Therefore, we conclude that, under the given evaluation criteria, it is recommended to apply all three processing techniques due to the reduction in vocabulary size. However, it should be noted that this also has an effect on other properties of the system, e.g. the quality of search results, which might or might not be desired.

6. SEARCH

The document index can now be used to retrieve the most relevant documents to a given query, which may be a combination of keywords and phrases. Those search capabilities are bundled in the Searcher class presented in Listing 5. The main search logic is contained within the *search* function as outlined in Listing 6, whereas phrase queries are processed using *extract_pq*.

```
class Searcher():
  def __init__(self,index_dir,analyzer):
    directory = SimpleFSDirectory(File(index_dir))
    reader = DirectoryReader.open(directory)
    self.searcher = IndexSearcher(reader)
    self.analyzer = analyzer

  def search(self,query,adv_query,N=0):
    # perform search for given query

  def extract_pq(self,q,field,slop=0,boost=5):
    # extract phrases from given query
```

Listing 5: Searcher class

```
def search(self,query,adv_query,N):
  bq = BooleanQuery()
  if query != '':
    # handle phrases in standard search
    pq,query = extract_pq(query,'content')
    if pq is not None:
      bq.add(pq,BooleanClause.Occur.MUST)
  if query != '':
    # handle remaining keywords
    qparser = QueryParser(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 = extract_pq(query,field)
      if pq is not None:
        bq.add(pq,BooleanClause.Occur.MUST)
      if query != '':
        # handle remaining keywords
        qparser = QueryParser(VERSION,field,
          self.analyzer)
        q = qparser.parse(query)
        bq.add(q,BooleanClause.Occur.MUST)
  return self.searcher.search(bq,N).scoreDocs
```

Listing 6: Query handling

Keyword search Our search allows free text keyword queries on any combination of the attributes title, authors, year and venue. By default, all attributes are considered. This is achieved by utilizing the *content* field of the index. We refer to this as a standard search (Listing 6, lines 8-13). Moreover, 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 (Listing 6, lines 21-26). Note that keywords are handled after phrase extraction (described below). Standard and advanced search can be combined.

Phrase queries Phrase queries are supported using double quotation marks (e.g. “information retrieval”) in both standard and advanced search (Listing 6, lines 3-7 and 17-20) through the *extract_pq* function outlined in Listing 7. There is no restriction for the number of phrases within a query. Documents containing a particular phrase receive higher weight during scoring. This requires an exact match up to the processing on the respective field (e.g. stemming). 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. All those rules are compactly represented using the query language of Lucene

(Listing 7, lines 11-12). Finally, the function returns the constructed sub-query as well as the remaining keywords.

```
1  def extract_pq(self,q,field,slop=0,boost=5):
2    phrases = re.findall(r'"([^"]*)"',q)
3    if len(phrases) == 0:
4      return None,q
5
6    q = re.sub(r'"([^"]*)"',"",q).strip()
7    bq = BooleanQuery()
8    for phrase in phrases:
9      qparser =
       QueryParser(VERSION,field,self.analyzer)
10     # handle phrase and single keywords
11     pq = qparser.parse('%s "%s"~%d^%.1f' %
       (phrase,phrase,slop,boost))
12     bq.add(pq,BooleanClause.Occur.MUST)
13   return bq,q
```

Listing 7: Phrase extraction

Query results 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.

7. EVALUATION

8. 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[7] to achieve responsive web design. Figure 1 depicts the search results for an example query. As the UI is not our focus, we omit further implementation details; the interested reader is referred to the source code.

9. APPLICATIONS IN IR

We implement two applications based on dblp data. The first one is to find the top-10 most popular research topics of a specific year. The second one is finding top-10 similar publication venues and years with a given publication venue and year. Only paper titles are used in both application.

In this section, the related techniques and the detailed implementations of the applications are demonstrated, followed by the experiment results.

9.1 Popular research topics

Before mining popular research topics from paper titles, we need to firstly extract topics from such data. In our application, we identify meaningful topics from titles based on the patterns of words. More precisely, given a query (e.g. 2014), we have following process to extract topics from it:

- Retrieve the titles of all the papers published in 2014, using the index.
- Tokenize each title and remove punctuations and stop words.
- Use Part-Of-Speech Tagger (POS Tagger)[8] to assign parts of speech to each word, such as noun, verb, adjective, etc.
- Extract topics from the tagged words based on the given pattern, using regular expression tools.

The key point of topic extraction is the above steps is to define the word pattern of topic. From the observation of real topic names, we can conclude that

- A topic name usually consist of adjectives and nouns.
- The adjectives always come before nouns for a topic name.
- Normally the number of words in a topic name is less than four.

Therefore, in the third step, we define the pattern as adj+adj+...+noun+... Moreover, only unigram, bigram and trigram are considered in this application. In addition, we ignore those unigram terms with very high (top 1%) frequency because they may be some widely-used terminology in many different topics. After the topic identification, we count the frequency of each topic appeared in the titles. Finally, the top-10 most frequent topics will be return as the result.

```
1  def get_popular_topics(self, q_year, top_k):
2      titles = self.searcher.search_year(q_year)
3      unigram_dist = {}
4      bigram_dist = {}
5      trigram_dist = {}
6      ngram_dist = {}
7
8      tagset = None
9      tagger = PerceptronTagger()
10     grammar = "NP:
       {<JJ>*(<NN>|<NNS>)*<NN>(<NN>|<NNS>)*}"
11     cp = nltk.RegexpParser(grammar)
12     for title in titles:
13         title = title.lower()
14         text = word_tokenize(title)
15         sentence = nltk.tag._pos_tag(text,
       tagset, tagger)
16         result = cp.parse(sentence)
17         for node in list(result):
18             if isinstance(node, nltk.tree.Tree):
19                 entity = zip(*list(node))[0]
20                 if len(entity) == 1:
21                     self.dict_append(entity,
       unigram_dist)
22                 elif len(entity) == 2:
23                     self.dict_append(entity,
       bigram_dist)
24                 elif len(entity) == 3:
25                     self.dict_append(entity,
       trigram_dist)
26                 else:
27                     self.dict_append(entity,
       ngram_dist)
28
29     unigram_result =
       Counter(unigram_dist).most_common(int(len(unigram_dist)
       * 0.01) + top_k)[int(len(unigram_dist) *
       0.01):]
30     bigram_result =
       Counter(bigram_dist).most_common(top_k)
31     trigram_result =
       Counter(trigram_dist).most_common(top_k)
32
33     result = unigram_result + bigram_result +
       trigram_result
34     result = sorted(result, key=lambda k: k[1],
       reverse=True)[:top_k]
```

[5]http://flask.pocoo.org
[6]http://jinja.pocoo.org
[7]http://getbootstrap.com
[8]http://nlp.stanford.edu/software/tagger.shtml

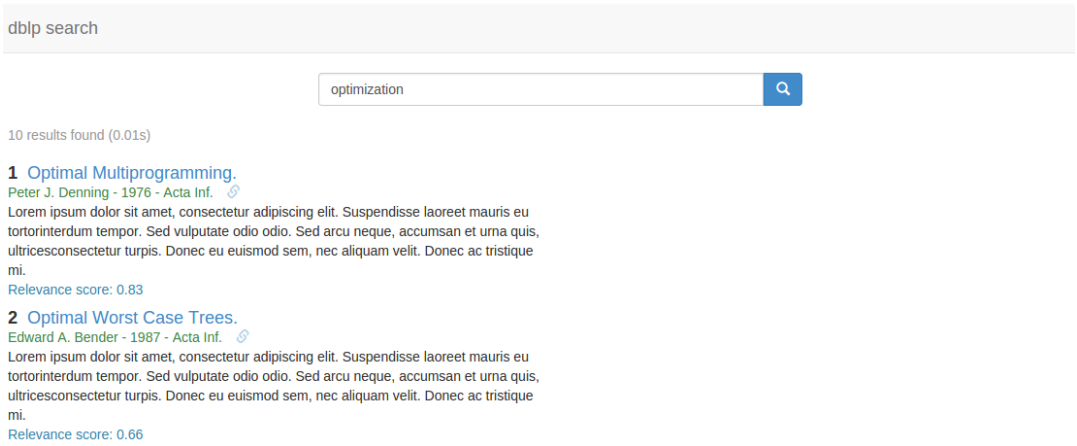

Figure 1: Web-based search user interface

```
35      return result
```

Listing 8: Finding popular topics for a given year

9.2 Similar publication venues

For retrieving similar publication venues and years, firstly we need to measure the similarity between two publication venues and years (e.g How similar are "TKDE'14" and "CIKM'15") using the paper titles. One simple way is to firstly tokenize every title of the venues and years, and use bag-of-words model, computing the cosine similarity of the word frequency vectors between them, which is defined as:

$$Sim_{i,j} = \frac{\sum_{k \in W} freq_{i,k} \cdot freq_{j,k}}{\sqrt{\sum_{k \in W} freq_{i,k}^2} \sqrt{\sum_{k \in W} freq_{j,k}^2}},$$

where i, j are two (venue, year) pairs. W refers to the collection contains all the words appeared in the corpus. $freq_{i,k}$ is the frequency of word k appears in the ith venue and year.

However, based on our statistical analysis, the dimensionality is too high. It is very inefficient to compute similarity. Besides, the data may also be very noisy, containing many words that only appears once in the whole dataset.

To address the aforementioned problems, we adopted Latent Dirichlet allocation (LDA)[1] to learn the latent topics from the data. Instead of using word frequency vectors, the similarity is defined based on the cosine similarity between the topic distribution of two venues and years. Thus, for a query venue and year, we compute the topic similarities between it and every other venues and years, which is given by

$$Sim_{i,j} = \frac{\sum_{z \in Z} \theta_{i,z} \cdot \theta_{j,z}}{\sqrt{\sum_{z \in Z} \theta_{i,z}^2} \sqrt{\sum_{z \in Z} \theta_{j,z}^2}},$$

where Z is the topic set; $\theta_{i,z}$ is the probability of the ith venue and year belongs to topic z.Then, based on the similarity, top-10 similar venues and years are returned as the result.

9.3 Application Experiments

For retrieving popular research topics, we have the results as follows:

Table 2: Top-10 most popular research topics of the papers published in 2013

Rank	Topic	Frequency
1	wireless sensor networks	1412
2	case study	1182
3	performance analysis	568
4	cognitive radio networks	468
5	special issue	452
6	performance evaluation	404
7	cloud computing	373
8	genetic algorithm	341
9	empirical study	340
10	neural network	340

Table 3: Representive words for the topics leant by LDA

Topic	Words
Geographic Information System	data using based sar spatial sensing analysis remote images radar model land image satellite surface detection gis classification resolution hyperspectral
Robotics	robot control based using robots mobile motion robotic planning human autonomous design navigation vision dynamic sensor multi force tracking visual
Wireless Sensor Networks	networks wireless based network mobile routing ad hoc sensor protocol performance control using traffic efficient scheme qos algorithm 802 multi
Software Technology	software testing analysis based using test engineering development study code case model systems java quality source empirical tool approach program
Social Media	social online media networks community network behavior use internet effects study analysis communities understanding games self role game impact influence

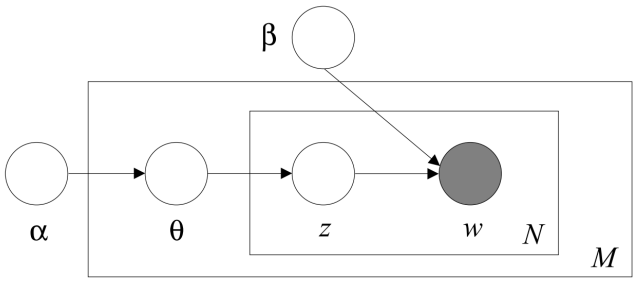

Figure 2: Graph representation of Latent Dirichlet allocation (LDA)

Table 4: Top-10 most similar venues & years to "IEEE Trans. Knowl. Data Eng. 2014"

Rank	Venue	Year	Similarity
1	IEEE Trans. Knowl. Data Eng.	2015	0.9898
2	IEEE Trans. Knowl. Data Eng.	2012	0.9888
3	DaWaK	2015	0.9887
4	IEEE Trans. Knowl. Data Eng.	2013	0.9856
5	LD4IE@ISWC	2014	0.9850
6	CIKM	2006	0.9845
7	WAIM	2013	0.9830
8	CIKM	2007	0.9823
9	FQAS	2009	0.9815
10	CIKM	2004	0.9803

For finding similar publication venues and years, we firstly learn the topics as follows:

The results of the queries are given as:

10. CONCLUSIONS

11. REFERENCES

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

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