Citation Search Engine

Academic paper search engine

Master Thesis

Aliya Ibragimova
University of Fribourg

Faculty of Natural Sciences
University of Bern

January 2015

Prof. Dr. Oscar Nierstrasz
Mr. Haidar Osman, Mr. Boris Spasojevic

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Nowadays the amount of documents in World Wide Web grows exponentially. Tools that can facilitate information retrieval present a particular interest in the modern world. A typical web search engine that search for information in World Wide Web is a software system that performs full-text indexing without considering meta information. This paper is devoted to the design of the academic paper search engine that takes advantage of meta-information, specifically citations. It is believed that citation is a very concise statement describing the source it refers to. Retrieving such statements can be particularly useful in writing scientific papers, for example, to build up a good argument. This paper describes implementation of Citation Search Engine, a system that makes an attempt to automatically extract, index and aggregate citations from a set of scientific articles in PDF format. Besides it analyses the results of the deployment of the system on the collection of scientific papers provided by Software Composition Group.

Contents

1 Introduction

1.1 Thesis statement

We believe that considering meta information helps to build enhanced search systems that can facilitate information retrieval. Particularly, we target information retrieval for scientific papers. We consider citations as important text blocks summarising or judging previous scientific findings assisting in creating a new scientific work. We propose Citation Search Engine a software system that extracts citations from scientific papers, aggregates citations based on the referred source, then indexes extracted content. It provides a practical web interface that allows users to search for citations.

1.2 Goals

We set following goals:

- Introduce the state of the art techniques in information search.
- Explore the structure of scientific articles, reveal common patterns
- Design and implement the academic search engine.
- Deploy the system on the given collection of scientific papers
- Analyse results, define future work

1.3 Outline

The rest of the paper structured as follows:

Chapter 2 The chapter first gives a high overview of the architecture of a typical web search engine and shortly reviews popular academic search engines. Then It describes the main steps for inverted index construction.

Chapter 3 Describes the design of Citation Search Engine. It first shows overall architecture of the proposed system and then describes details of implementation of each component.

Chapter 4 The chapter describes the deployment process and analysis the result of setting up the system on the given collection of scientific articles.

Chapter 5 Evaluation

Chapter 6 Contains conclusion and possible future work.

2
Related Work

2.1 Typical web search engine

Figure 2.1 illustrates a high level architecture of a standard web engine. It consist of three main components:

- Web Crawler
- Data indexer
- Search interface

Figure 2.1: A high-level architecture of a typical web search engine

Web Crawler is a program that browses the World Wide Web reading the content of web pages in order to provide up-to-date data to Data Indexer. Data Indexer decides how a page

content should be stored in an index database. Index helps to quickly query information. Users can search and view query results through Search Interface. When user makes a query the search engine analysis its index and returns best matched web pages according to specific to indexer criteria.

Web crawlers that fetch web pages with the content in the same domain are called focused or topical crawlers. An example of focused crawlers are academic-focused crawlers that crawls academical documents. Such crawlers become components of the "focused" search engines. Next chapter reviews some of popular academical search engines.

2.2 Popular academic search engines

CiteSeer^x CiteSeer^x is an autonomous citation search engine [4, 10]. CiteSeer^x automatically parses and index publicly available scientific articles found on the World Wide Web. It uses the impact of citations to rank documents. CiteSeer^x is built on the open source infrastructure SeerSuite [11] and uses Apache Solr [3] search platform for indexing documents. It can extract meta information from papers such as title, authors, abstract, citations. The extraction methods are based on machine learning approaches such ParseCit [1]. CiteSeer^x one of the world's top repositories and was rated number 1 in July 2010 [7]. It currently has over 4 million documents with nearly 4 million unique authors and 80 million citations. CiteSeer^x focuses on indexing citations more precisely bibliographic links while in Citation Search Engine we intend to index text of the citations in a body of a document.

Google Scholar Google Scholar is a freely accessible web search engine that makes full-text indexing of scientific literature [5]. Among features of Google Scholar engine are unique ranking algorithm, "cited by" feature, allowing to view abstracts of the articles citied the given article, "related articles" feature, showing the list of closely related articles. Google Scholar contains roughly 160 million of documents by May 2014 [8].

2.3 Inverted index

Search engines like CiteSeer or Google Scholar deal with a large collection of documents. When user makes a query to such systems one would like to have a mechanism to process document collections quickly. The way to avoid linear scanning the text of all documents for each query is to *index* them in advance. Thereby we are coming to the concept of *inverted index* that is the major concept in information retrieval. The term *inverted index* comes from the data structure storing a mapping from content, such as words or numbers, to the parts of a document where it occurs. Figure 2.2 shows the basic idea of an inverted index. We have a dictionary of terms appearing in the documents. Each term maps to a list that records which documents the term occurs in. Each item in the list, conventionally named as *posting*, records that a term appears in a document, often it records the position of the term in the document as well. The dictionary on Figure 2.2 has been sorted alphabetically and each postings list is sorted by document ID. Document ID is a unique number that can be assigned to a document when it's first encountered.

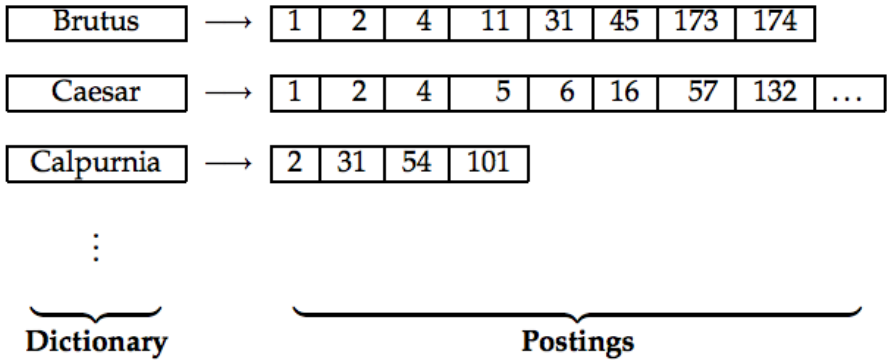

Figure 2.2: Inverted index example

The construction of the inverted index has following steps:

1. Documents collection
2. Breaking each document into tokens, turning a document into a list of tokens
3. Linguistic preprocessing of a list of tokens into normalised list of tokens
4. Index documents by creating an inverted index, consisting of a dictionary and postings

First step of the index construction is documents collection that aims to obtain a set of documents containing sequence of characters. Usually the input to the indexing process is digital documents that are bytes in a file or a web server. While for the plain English text in ASCII encoding solution is straightforward there might be trickier cases. Consider for example a collection of PDF files, we need to correctly decode out characters of some binary representation. Finally, the textual part of the document may need to be extracted out of other parts that will not be processed. Next we should determine a document unit, for example it might be a chapter in a book, or a paragraph in a scientific article.

After getting the sequence of characters in document units next step is to breaking up documents into *tokens*. Token can be think of a semantical unit for processing, for example, it might be a word or a number. At the same time during tokenisation some characters like punctuations can be thrown out.

Here is a tokenisation example:

Input: Friends, Romans, Countrymen, lend me your ears;
Output: Friends | Romans | Countrymen | lend | me | your | ears

The third step is normalisation. It's good when tokens in a user query match tokens in the token list of documents. Consider an example of querying the word *co-operation*, a user might also be interested in getting documents containing *cooperaion*. *Token normalisation* is a process

of turning a token into a canonical form so matches can occur despite superficial differences in the character sequences. One way of token normalisation is keeping relations between unnormalised tokens, which can be extended to manual constructed synonym lists, such as *car* and *automobile*. The most standard way of token normalisation however is creating *equivalence classes*. If tokens become identical after applying a set of rules then they are the equivalence classes. Consider some common normalisation rules that are often used:

Stemming and lemmatisation Words can be used in different grammatical forms, for instance, *organize*, *organizes*, *organizing*, however in many cases it sounds reasonable for one of these words to return documents contain other forms of the word. The goal of stemming and lemmatisation is reduce a form of the word to a common base form.

Here is an example:

am, are, is = >be

car, cars, car's, cars' =>car

The result of applying the rule to the sentense:

the boys cars are different colors =>the boy car be differ color

Stemming and lemmatisation are closely related concepts however there is a difference. *Lemmatisation* usually refers to finding a *lemma*, common base of a word, with the help of a vocabulary, morphological analysis of a word and requires understanding the context of a word and language grammar. *Stemming* however operates with a word without knowing it context and thereby can't distinguish the same words having different meanings depending on the context.

Here is an example:

better = >good , can only be matched by lemmatisation since requires dictionary look-up

writing =>write, can be matched by both lemmatisation and stemming

meeting =>meeting(noun) or to meet(verb), can be matched only by lemmatisation since requires the word context

In general stemmers are easier to implement and run faster. The most common algorithm for stemming is *Porter's* algorithm [9].

Capitalization/case-folding A simple strategy is to reduce all letters to a lower case, so that sentences with *Automobile* will match to queries with *automobile*., however this approach would not be appropriate for example to identifying company names, such as *General Motors*. The better strategy for English language would be to lowercase words only in the beginning of the sentences and to lowercase all words in titles. Case-folding can be be done more accurately by a machine learning model using more features to identify weather a words should be lowercased.

Accents and diacritics Diacritics in English language play an insignificant role and simply can be removed. For instance *cliché* can be substituted by *cliche*. In other languages diacritics can be part of the writing system and distinguish different sounds. However in many cases

users can enter queries for words without diacritics, whether for reasons of speed, laziness or limited software

The last step is the core part of the building inverted index. The input to indexing is a list of normalised tokens for each document, which is a list of pairs of term and document ID, as on Figure 2.3. The indexing algorithm is sorting the input list so that the terms are in alphabetical order. Then it merges the same terms from the same document. And finally instances of the same term are grouped and the result is split into a dictionary and postings, as shown on Figure 2.2.

term	docID	term	docID
I	1	ambitious	2
did	1	be	2
enact	1	brutus	1
julius	1	brutus	2
caesar	1	capitol	1
I	1	caesar	1
was	1	caesar	2
killed	1	caesar	2
i′	1	did	1
the	1	enact	1
capitol	1	hath	1
brutus	1	I	1
killed	1	I	1
me	1	i′	1

Figure 2.3: Input to the indexing algorithm

2.4 Dynamic indexing

So far we assumed that document collection is static however there are many cases when collection can be updated, for example, by adding new documents, deleting or updating existing documents. One simple way dealing with dynamic collection is to reconstruct the inverted index from scratch. This might be acceptable if changes made in collection are small over time and delay in making new documents searchable is not critical. However if there is one of a requirement mentioned above, for example, making new documents searchable quickly then one might be interested in another solution: keeping auxiliary index. Thus we have a large main index and we keep auxiliary index for changes. The auxiliary index is kept in memory. Every time a user makes a query the search runs over both of the indexes and results are merged. When auxiliary index becomes too large it can be merged with the main index.

2.5 Retrieving search results

When a user makes a query she would be interested in getting a result document containing all terms in the query so that terms are close to each other. Consider an example of querying a

phrase containing 4 terms. The part of the document that contains all terms is named a *window*. The size of the window is measured in a number of words. For instance a smallest window for 4-terms query will be 4. Intuitively, smaller windows represent better results for users. Such window can become one of the indicators scoring a document in the search result. If there is no document containing all 4 terms, a 3-term phrase can be queried. Usually search systems hides the complexity of searching a result from user introducing *free text query parsers* so a user can make only one query.

2.6 Conclusion

Figure 2.4 summarises approaches described above in a more detailed picture of a basic search system. Left stream in the Figure 2.4 describes the process of parsing a set of documents and applying linguistic processings (tokenisation and lemmatisation) in order to built indexes with the help of a indexer. The middle stream on Figure 2.4 represents a user making a query, where free text parsers together with spell checkers send requests to the index bank. The index bank returns document candidates to a scoring and ranking component, the left stream on Figure 2.4. Finally, ranked document are shown to the user as a result page.

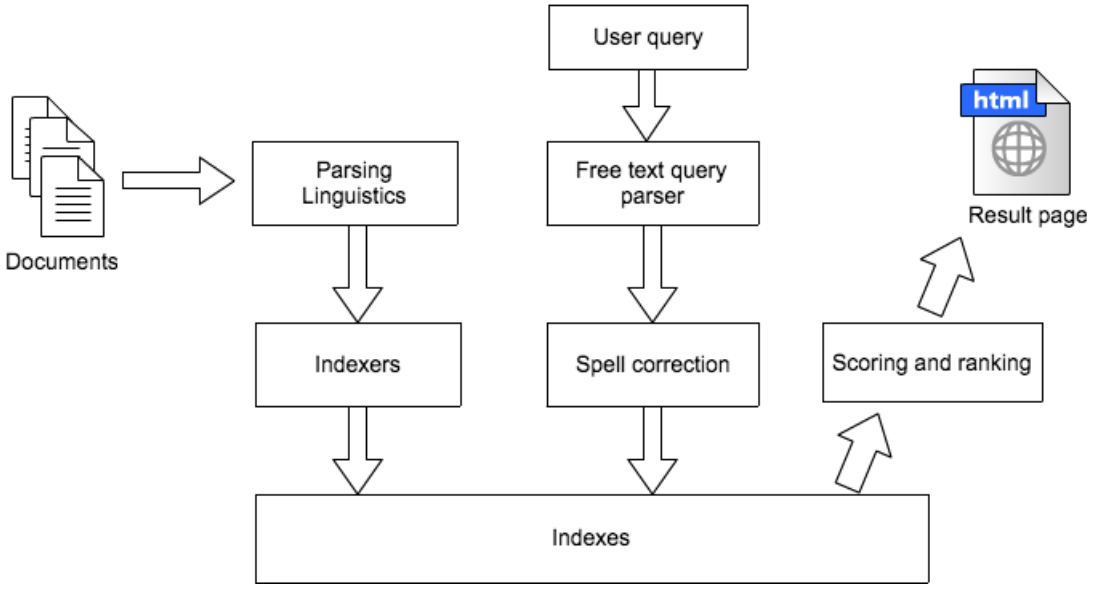

Figure 2.4: Search Engine internals

3
Citation Search Engine

3.1 System overview

The overall architecture of Citation Search Engine is shown on Figure 3.1. There are three main operations performed by Citation Search Engine: parsing PDF files, indexing documents and querying on resulted indexes. Correspondingly there are three major components responsible for accomplishment of these operations: *Parser*, *Indexer Solution* and *Search Web App*. The system has two more components for storing data: *Indexes Storage* for storing indexes and *Meta Information Storage* for storing meta information. For the convenience, the workflow of the system is numbered and will be explained below.

First operation performed by the system is indexing. In the indexing process collection of PDF files is provided to *Parser* (1). *Parser* parses PDF files into text and extract meta information from text. It then packs data about citations into documents and publishes obtained documents to *Indexer Solution* (2) which indexes and stores indexes in *Indexes Storage* (2'). Meta information extracted from textual representation of PDF files that doesn't require indexing is stored in *Meta Information Storage* (3).

In the querying process user searches for some phrase using *Search Web App* user interface. The phrase is analysed by *Indexer Solution*. *Indexer Solution* finds documents matching the query phrase in *Indexes Storage* (2') and sends them to *Search Web App* (4). If user is interested in getting details regarding the specific reference the information will be looked up in *Meta Information Storage* (5). Finally, the result is shown to the user as a an html page (6).

Next sections of the chapter describe implementation of each component in detail and show the reasons of choosing a particular solution.

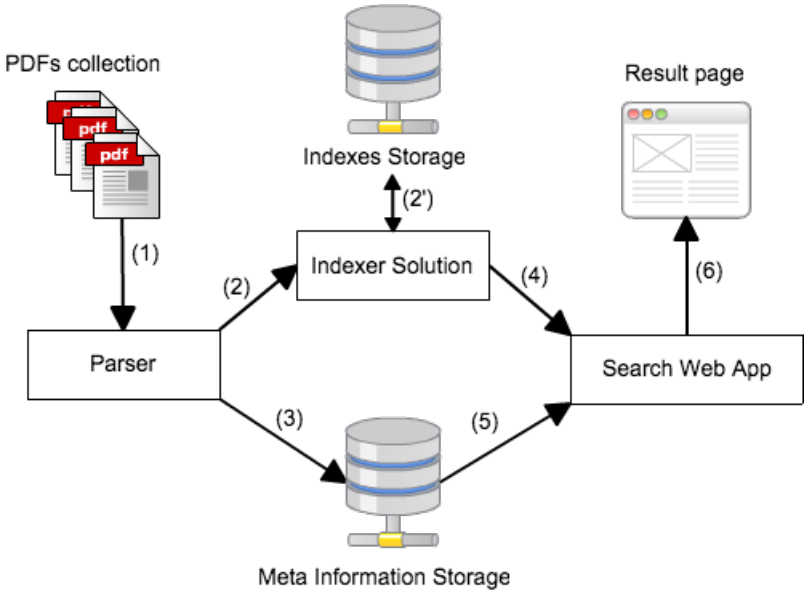

Figure 3.1: Search Engine high level architecture

3.2 Parser

It is practical to divide the work of *Parser* into two phases: *Processing* and *Publishing*, as on Figure 3.2. The output of *Processing* phase is input to *Publishing* phase. *Processing* phase aims to parse PDF-files and prepare documents with extracted information for publishing. Next parts of this section describes *Processing* and *Publishing* phases in detail.

Figure 3.2: Parser workflow

3.2.1 Processing

The main role of *Processing* phase is to parse scientific articles into text and extract citations and bibliographic references to create documents for publishing. In common case parsing PDF-files from different sources is a very challenging task as there is a large number of scientific articles in different formats. Besides that there exist scientific articles written decades ago which can use different encoding algorithms than articles created recently. Thereby building a universal parser is practically impossible. In our case we try to identify common patterns covering majority of scientific article formats or at least formats found in our collection of articles.

Processing phase starts with recursively walking though directory tree of the collection library. While walking though directory *Parser* filters non-pdf files and parses and processes each PDF-file separately. We used Apache PDFBox library [2] to parse PDF-files into text. The library extracts full text from PDF-file but without any hints on the initial structure of the article. In our case to find citations and bibliographic references in text we need to look them up in different parts of the article, so it was designed an algorithm to break PDF-text into sections.

Generally we are interested in identifying a body of the document where we can find citations and references block where we can find bibliographic links. One way of finding these sections can be using keywords that might signify the beginning or the end of some section. Based on keywords on can extract different sections of a document. Figure 3.2 shows a sample text of a parsed pdf-document with keywords.

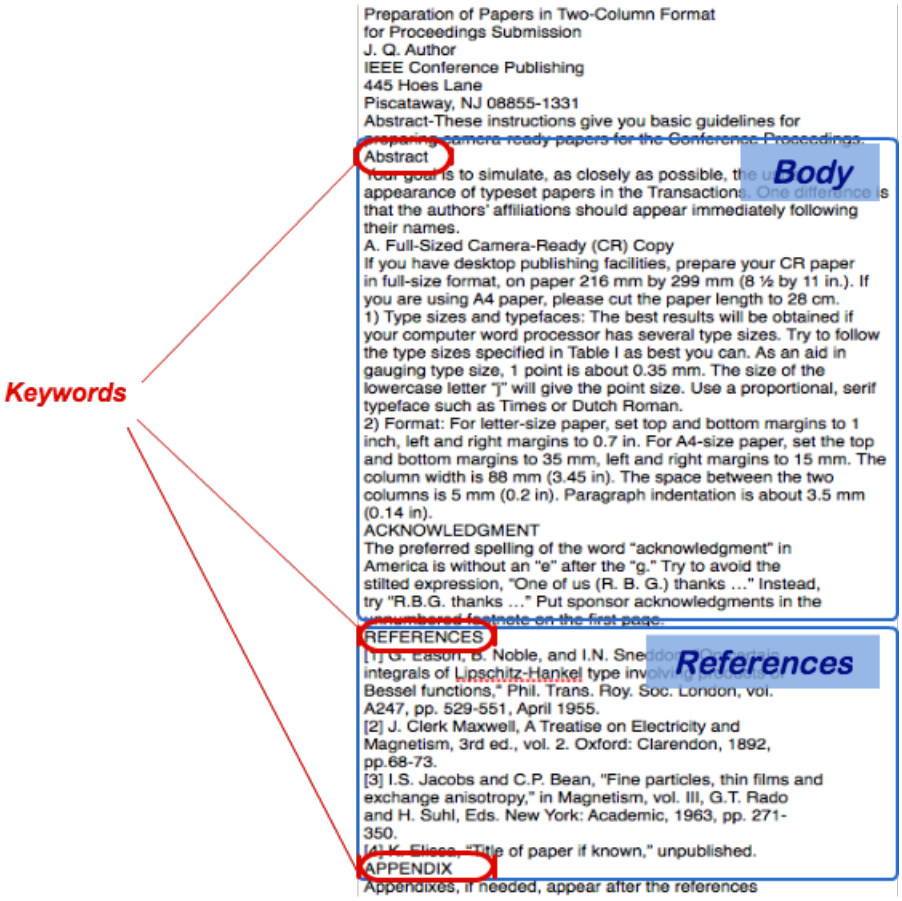

Figure 3.3: Sample text of the parsed scientific article. Keywords help to break the document into sections.

One can notice following characteristics of scientific articles:

- Body of a document comes before references section

- Appendix or Author's biography sections can come after references section
- Each document contains an Abstract and References words and might contain Appendix word. We call these words keywords.

In common case keywords can be written in different formats, for example, using upper or lower case. Table 3.1 illustrates variations of keywords in the beginning of document sections.

body	references	appendix
Abstract	References	Appendix
ABSTRACT	References:	APPENDIX
	REFERENCES	

Table 3.1: Keywords identifying different sections in a document

After breaking down a document into sections as show on Figure 3.3, text in sections are presented in one-column format. There are two aspects regarding this format. First, sentences can be splitted by new line symbol in the end of a line. Secondly, words can be splitted by dash symbol in the end of a line. It was introduce a normalization step where new lines are substituted by white spaces and dashes are removed in the end of a line to obtain continues text.

As a result of normalization step we have a document divided into body and references section. Before searching citations in the body of a document it is reasonable to break body into sentences. In general, breaking text into sentences is not an easy task. Consider a simple example with a period. Period not only indicates the end of a sentence but also can encounter in a sentence itself, for example, an item of a numbered list or a name of a scientist. Besides, not all sentences end by period, for example, title of a section or an item of a list. It was used Stanford CoreNLP library that employs natural language and machine learning techniques to extract sentences [6].

Next, we search for citations in a body of a document and bibliography links in a references section. When an author makes a citation it puts a link to a bibliographic reference in the sentence. It is common to use square brackets ([and]) to make a link to a bibliographic reference in the sentence. Thus, detecting square brackets in the text we can identify citations. After analyzing some set of articles it was revealed frequent patterns in using square brackets for citing showed in Table 3.2.

Further we need to extract bibliographic links from a references section. For that we studied most common variants of composing references sections. Table 3.3 surmises these findings. To extract bibliographic links we make regular epressions matching one of those patterns listing in Table 3.3. Extracting bibliographic links allow us to match citations with bibliographic links.

The pipline of processing stage described above depicted on Figure 3.4. As seen from Figure 3.4 to complete the processing stage we need to perform one more step: extracting titles from bibliographic links.

3.2.2 Publishing

There are two systems where documents will be published: Solr and MongoDB. As already explained above Solr is used for indexing citations and MongoDB for storing meta-data. Some

Patterns of using [ ]	Example in text
[21]	Our conclusion is that, contrary to prior pessimism **[21]**, [22], data mining static code attributes to learn defect predictors is useful.
[20, 3, 11, 17]	In the nineties, researchers focused on specialized multivariate models, i.e., models based on sets of metrics selected for specific application areas and particular development environments **[20, 3, 11, 17]**.
[24, Sections 6.3 and 6.4]	Details on the life-cycle of a bug can be found in the BUGZILLA documentation **[24, Sections 6.3 and 6.4]**.
[PJe02]	In a lazy language like Haskell **[PJe02]** this is not an issue - which is one key reason Haskell is very good at defining domain specific languages.

Table 3.2: Frequent patterns in using square brackets ([and])for citing

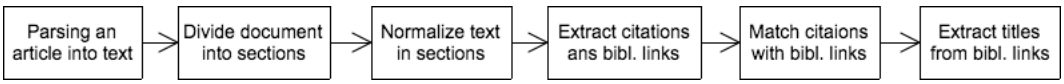

Figure 3.4: Pipeline of processing stage

reasonings in favor of using MongoDB additionally to Solr storage will be given below.

3.2.3 Publishing to Solr

The common way to interact with Solr is using REST API. Solr provides client libraries for many programming languages to handle interactions with Solr's REST API. In our project we used SolrJ client library for Java language. This library abstract interaction with Solr into java objects instead of using typical XML request/response format. Basic Solr storage unit is called document and SolrJ has document abstraction implementation. For every detected citation we compose a document to publish. Figure 3.5 represents a structure of documents we publish to Solr.

Every document representing one citation consist of following fields:

- id: document unique id, mandatory field for publishing to Solr
- text: text of the citation that we want to index
- context: citation with a text framing it, we take 1 sentence before and 1 after the citation
- path: URL of a document where citation was found
- references: list of bibliographic links from references section matching this citation

Lets have a look at how Solr stores documents. Solr uses NoSQL like data storage, but actually it is even more limited than traditional NoSQL databases. Indeed, the data stored in Solr

References section templates
[1] J. Bach. Useful features of a test automation system (partiii) . . . **[2]** B. Beizer. Black-Box Testing. John Wiley and Sons, ...
1. J. R. Hobbs, Granularity, Ninth International Joint Conference . . . **2.** W. Woods, What's in a Link: Foundations for Semantic Networks, ...
[1]. Arnold, R.S., Software Reengineering, ed. . . . **[2].** Larman, C., Applying UML and Patterns. 1998, ...
[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: . . . **[AU73]** A.V. Aho and J.D. Ullman. The theory of parsing, translation . . . . . .

Table 3.3: Frequent patterns of writing references block

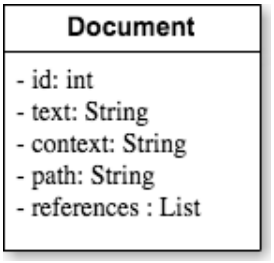

Figure 3.5: Document structure publishing to Solr

storage is very 'flat', which means that Solr doesn't allow to store even hierarchical data. In our case along with the references we intent to store a title of the scientific article parsed from the reference string, so later we can aggregate citations referred to the same scientific article. /*It's worth to say that citation itself can refer to multiple scientific articles*/ We are also interested in a solution that doesn't require reviewing all Solr documents to find citations referred to the same scientific article as it will be too slow and will decrease quality of user experience. Thus we are coming to the external storage solution that will keep titles of scientific articles and all citations referred to a specific article. As there are few relations in our data and we would like to have a scalable solution it was decided to use MongoDB as an external storage.

3.2.4 Publishing to MongoDB

MongoDB is a document-oriented NoSQL database that stores data in JSON-like documents with dynamic schema. To connect to database we used Java driver provided by MongoDB. Although MongoDB is a 'schemaless' database we adhere to JSON structure of a document showed on Listing 1. The json document consist of following fields:

- id: document id, field automatically assigned by MongoDB
- title: title of a scientific article
- cittations: citations with its references of the scientific article identifying by title field

Every time while sending a new citation with paper title to MongoDB we check if document with the same title already exist. If so we add a new citaion to this document otherwise create a new document.

```
{
   "_id" : ObjectId("547ef1b219795f049d6a0ad0"),
   "title" : "Re-examining the Fault Density-Component Size  Connection",
   "citations" : [
   {
      "citation" : "Hatton, [19], claims that there is compelling empirical
                 evidence from disparate sources to suggest that in any
                 software system, larger components are proportionally more
                 reliable than smaller components.",
      "references" : [
      "[19] L. Hatton, Re-examining the Fault Density-Component Size ..."
      ]
   },
   {
      "citation" : "Hatton examined a number of data sets, [15], [18] and
                 concluded that  there was evidence of macroscopic behavior
                 common to  all data sets despite the massive internal
                 complexity of each  system studied, [19].",
      "references" : [
      "[15] K.H. Moeller and D. Paulish, An Empirical Investigation of ...",
      "[18] T. Keller, Measurements Role in Providing Error-Free Onboard ...",
      "[19] L. Hatton, Re-examining the Fault Density-Component Size  ..."
      ]
   }]
}
```

Listing 1: Sample document stored in MongoDB

4
Validation

In which you show how well the solution works.

5
Conclusion and Future Work

In which we step back, have a critical look at the entire work, then conclude, and learn what lays beyond this thesis.

Bibliography

[1] Isaac G. Councill, C. Lee Giles, and Min yen Kan. Parscit: An open-source crf reference string parsing package. In *INTERNATIONAL LANGUAGE RESOURCES AND EVALUATION*. European Language Resources Association, 2008.

[2] The Apache Software Foundation. Apache PDFBox. https://pdfbox.apache.org/.

[3] The Apache Software Foundation. Apache Solr. http://lucene.apache.org/solr/.

[4] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries*, DL '98, pages 89–98, New York, NY, USA, 1998. ACM.

[5] Google. Google Scholar. http://scholar.google.ch/.

[6] The Stanford Natural Language Processing Group. Stanford CoreNLP API. http://nlp.stanford.edu/software/corenlp.shtml.

[7] Cybermetrics Lab. Ranking Web of Repositories. http://repositories.webometrics.info/.

[8] Enrique Orduña Malea, Juan M. Ayllón, Alberto Martín-Martín, and Emilio Delgado López-Cózar. About the size of google scholar: playing the numbers, July 2014.

[9] M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[10] The Pennsylvania State University. Citeseer. http://citeseerx.ist.psu.edu/.

[11] The Pennsylvania State University. SeerSuite. http://citeseerx.sourceforge.net/.