Citation Search Engine

Academic paper search engine

Master Thesis

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 number of documents in the World Wide Web grows extremely fast. Tools that can facilitate information retrieval present a particular interest in the modern world. We believe that considering meta information helps to build enhanced search systems that can facilitate information retrieval. Particularly, we target information retrieval for scientific articles. We consider citations in scientific articles as important text blocks summarizing or judging previous scientific findings, assisting in creating a new scientific work.

We propose Citation Search Engine a software 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 a collection of scientific papers. It evaluates searching capabilities of the system by comparing with alternative approaches.

Contents

1 Introduction

1.1 Thesis statement

Searching over large collection of documents by hand is time inefficient. The common way of making search effective in terms of time is indexing documents in advance. Before indexing procedure, documents should be parsed and analyzed. In this work we aim to build search system over collection of scientific articles in PDF format. Scientific articles are different from ordinary documents in following sense. First, scientific articles in PDF need to be converted into textual representation. Secondly, scientific articles have a certain layout format, where each section need to be considered separately. In Citation Search Engine, we do not aim to index full text of articles, instead we aim to take advantage of indexing extracted meta-information. We believe that considering meta-information might enhance a search system. We particularly focused on extracting citations from scientific articles. In general, a citation composed of two parts: a cited text in the body of a document and a bibliographic link in a reference section of a document. In Citation Search Engine we index both parts separately allowing user to make a choice between different indexed collections. We also store context of a citation to better understanding of the idea of the cited text. Citation context is a text framing cited text. Bibliographic links identifies the cited origin paper. By finding bibliographic links pointing to the same original source we can merge citations related to the same paper. Such functionality is especially useful to automatically

find citations of a given article in other articles.

1.2 Highlights

Following are highlights of this work:

- Design and implement the Citation Search Engine.
- Deploy the system on the given collection of scientific papers
- Make evaluation by comparing with alternative approaches
- Analyze results, define future work

1.3 Outline

The rest of the paper structured as follows:

Chapter 1 The chapter Gives a high overview of the architecture of a typical web search engine. It describes the main steps to construct an inverted index.

Chapter 2 The chapter surveys the research related to citations in scientific publications. It makes an overview of two popular academic search engines: Google Scholar and CiteSeer.

Chapter 3 The chapter describes the design of Citation Search Engine. It first shows overall architecture of the proposed system and then shows 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 5 Contains conclusion and possible future work.

1.4 Typical web search engine

Figure 1.1 illustrates a high level architecture of a standard web engine. It consists of three main components:

- Crawler
- Indexer
- Index Storage
- Search interface

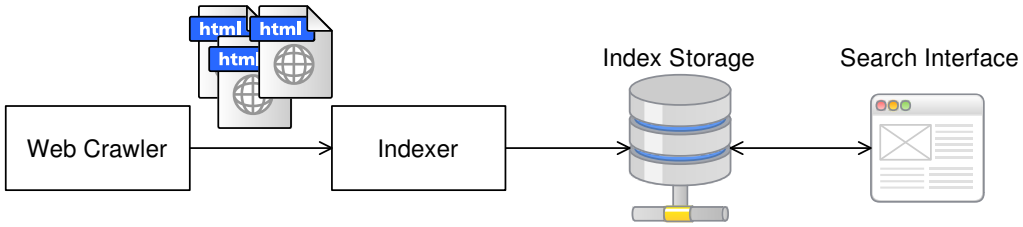

Figure 1.1: A high-level architecture of a typical web search engine

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 Indexer. Indexer decides how a page content should be stored in an index storage. Indices help to quickly query documents from the index storage. Users can search and view query results through the Search Interface. When a user makes a query the search engine analyzes its index and returns best matched web pages according to specific criteria.

Web crawlers that fetch web pages with the content in the same domain are called focused or topical crawlers. An example of a focused crawler is an academic-focused crawler that crawls scientific articles. Such crawlers become components of focused search engines. Examples of popular academic search engines are Google Scholar and CiteSeer. Chapter 2 gives an overview of these search engines.

1.5 Inverted index

Search engines like CiteSeer or Google Scholar deal with a large collection of documents. 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*, which is a 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 1.2 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 1.2 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.

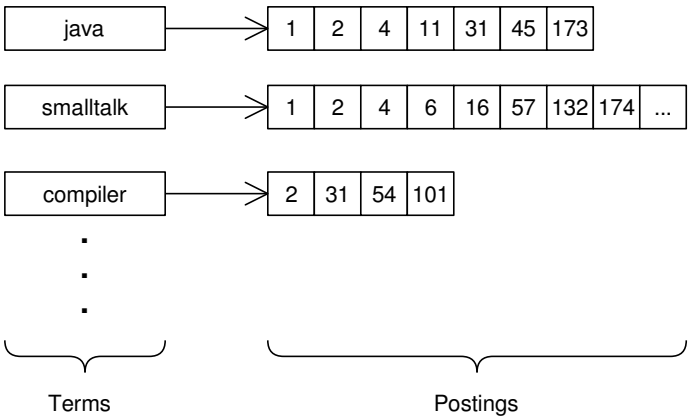

Figure 1.2: Example of an inverted index. Each term in a dictionary maps to a posting list consisting of document IDs, where this term occurs. Dictionary terms are sorted alphabetically and posting lists are sorted by document IDs

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 with terms and postings

First step of the index construction is documents collection. It aims at obtaining a set of documents containing a sequence of characters. The input to the indexing process could be digital documents that are bytes in a file or a web server. Consider, for example, a collection of PDF

files. First, we need to correctly decode out characters of the binary representation. Next we should determine a document unit. For example, it might be a chapter in a book, or a paragraph in a scientific article.

The next step after getting the sequences of characters in the document units, is to break up documents into *tokens*. Tokens can be thought of as the semantical units for processing. For example, it might be a word or a number. During tokenisation some characters like punctuations can be thrown out.

Here is a tokenisation example:

Input: Sometimes, I forget things.
Output: Sometimes | I | forget | things

The third step is normalization. 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 lexical differences in the character sequences. One way of token normalisation is keeping relations between unnormalized tokens, which can be extended to manual constructed synonym lists, such as *car* and *automobile*. The most standard way of token normalization however is creating *equivalence classes*. If tokens become identical after applying a set of rules then they are in the equivalence classes. Consider the following common normalization rules that are often used:

Stemming and Lemmatization 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 lemmatization is to reduce the 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 sentence:

three frogs are flying =>three frog be fly

Stemming and lemmatization are closely related concepts however there is a difference. *Lemmatization* 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 its context and thereby can't distinguish that the same words have different meanings depending on the context.

Here is an example:

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

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

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

In general, stemmers are easier to implement and run faster. The most common algorithm for stemming is *Porter's* algorithm [19].

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 in some contexts like identifying company names, such as *General Motors*. Case-folding can be be done more accurately by a machine learning model using more features to identify whether a word 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.

The last step of building the inverted index is sorting. The input to indexing is a list of pairs of normalized tokens and documents IDs for each document. Consider an example of three documents with their contents:

- Document 1: Follow the rules.
- Document 2: This is our town.
- Document 3: The gates are open.

After applying tokenasation and normalisation steps of the listed documents the input to the indexing is shown in Table 1.1. The indexing algorithm sorts the input list so that the terms are in alphabetical order as in Table 1.2. Then it merges the same terms from the same document by folding two identical adjacent items in the list. And finally instances of the same term are grouped and the result is split into a dictionary with postings, as shown in Table 1.3.

Term	DocumentID
follow	1
the	1
rule	1
this	2
be	2
our	2
town	2
the	3
gate	3
be	3
open	3

Table 1.1: Input to the indexing algorithm is a list of pairs of a term and document ID, where this term occurs.

1.6 Dynamic indexing

So far we assumed that document collection is static. However there are many cases when the collection can be updated, for example, by adding new documents, deleting or updating existing documents. Simple way to deal with dynamic collection is to reconstruct the inverted index from scratch. This might be acceptable if the changes made in the collection are small over time and the delay in making new documents searchable is not critical. However if there is one of the aforementioned conditions is violated, one might be interested in another more dynamic solution like keeping an 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 indexes and results are merged. When the auxiliary index becomes too large it can be merged with the main index.

1.7 Retrieving search results

When a user makes a query it would be good to give her back a result document containing all terms in the query, so that terms are located close to each other in the document. 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 number of words. For instance the smallest window for 4-term query will be 4. Intuitively, smaller windows represent better results for users. Such window can become one of the parameters ranking a document in the search result. If there is no document containing all 4 terms, a 3-term phrase can be queried.

Term	DocumentID
be	2
be	3
follow	1
gate	3
open	3
our	2
rule	1
the	1
the	3
this	2
town	2

Table 1.2: Indexing algorithm sorts all terms in a alphabetical order. The result is a list of sorted terms with document IDs

Term	Postings
be	2 3
follow	1
gate	3
open	3
our	2
rule	1
the	1 3
this	2
town	2

Table 1.3: Indexing algorithm groups the same terms with creating postings. The result is a dictionary with terms as keywords and values as postings.

Usually search systems hide the complexity querying from the user by introducing *free text query parsers* so a user can make only one query.

2
Related Work

2.1 Citations in scientific publications

Citations are the subject of many interesting scientific studies.

Bradshaw et al. [3] showed that citations provide many different perspectives on the same article. They believe that citation provide means to measure the relative impact of articles in a collection of scientific literature. In their work the authors improved the relevance of documents in the search engine results with a method called Reference Directed Indexing. Reference Directed Indexing (RDI) is based on a comparison of the terms authors use in reference to documents.

Bertin and Atanassova [1] [14] [2] automatically extract citations and annotate them using a set of semantic categories. In [14] and [1] they used linguistic approach, which used the contextual exploration method, to annotate automatically the text. In [2] they proposed a hybrid method for the extraction and characterization of citations in scientific papers using machine learning combined with rule-based approaches.

There are several studies that used citations to evaluate science by introducing map of science. Map of science graphically reflects the structure, evolution and main contributors of a given scientific field [6] [11] [13] [23].

Kessler [10] first used the concept of bibliographic coupling for document clustering. To build a cluster of similar documents Kessler used a similarity function based on the degree of

bibliographic coupling. Bibliographic coupling is a number of citations two documents have in common. The idea was developed further by Small in co-citation analysis [22]. Later co-citation analysis and bibliographic coupling was used by Larson [12] for measuring similarity of wwww pages.

Another approach is to use citations to build summaries of scientific publications. There are three categories of summaries proposed based on citations: overview of a research area (multi-document summarization) [17], impact summary (single document summary with citations from the scientific article itself) [15] and citation summary (multi- and single document summarization, in which citations from other papers are considered) [20]. In work by Nakov et al. citations have been used to support automatic paraphrasing [16].

An expert literature survey on citation analysis was made by Smith [24], she reviewed hundred of scientific articles on this topic.

2.2 Popular academic search engines

CiteSeer^{x} CiteSeer^{x} is built on the concept of citation index. The concept of citation index was first introduced by Eugene Garfield in [7]. In terms of Eugene Garfield citations are bibliographic links or referrers linking scientific documents. In his work Eugene Garfield proposed an approach where citations between documents were manually cataloged and maintained so that a researcher can search through listings of citations traversing citation links either back through supporting literature or forward through the work of later researchers [4].

Lawrence et al. automated this process in CiteSeer^{x} [1] [8], a Web-based information system that permits users to browse the citation links between documents as hyperlinks. CiteSeer^{x} automatically parses and indexes publicly available scientific articles found on the World Wide Web.

CiteSeer^{x} is built on top of the the open source infrastructure SeerSuite [2] 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 as ParseCit [5]. CiteSeer^{x} currently has over 4 million documents with nearly 4 million unique authors and 80 million citations.

CiteSeer^{x} indexes citations more precisely bibliographic links or referrers while in Citation Search Engine we intend to index not only bibliographic links but also cited text in a body of a document. If by indexing bibliographic links CiteSeer^{x} mainly aims to simplify navigation

[1] CiteSeer, http://citeseerx.ist.psu.edu/

[2] SeerSuite, http://citeseerx.sourceforge.net/

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

between linked documents, in Citation Search we focus on simplifying retrieval of documents containing a text of interest.

Google Scholar Google Scholar is a freely accessible web search engine that makes full-text or metadata indexing of scientific literature [4]. Besides the simple search Google Scholar proposes following features. Unique ranking algorithm, an algorithm that ranks documents "the way researchers do, weighing the full text of each document, where it was published, who it was written by, as well as how often and how recently it has been cited in other scholarly literature" [5]."Cited by" feature allows to view abstracts of articles citing the given article. "Related articles" feature shows the list of closely related articles. It is also possible to filter articles by author name or published date. Google Scholar contains roughly 160 million of documents by May 2014 [18].

[4]Google Scholar, http://scholar.google.ch/
[5]https://scholar.google.com/scholar/about.html

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~~→the resulted indexes. Correspondingly, there are three major components responsible for accomplishment of these operations: *Parser*, *Indexer Solution* H. ►*Indexer Solution is not a good name for a component. Just name it the Indexer*◄ 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 H. ►*First, the names are pretty self-explanatory. Second: Why there are two different storage? You have to justify.*◄. For the convenience, the workflow of the system is numbered and will be explained below.

H. ►*There should be two diagrams. One component diagram and one activity diagram. Then you should explain each one one its own.*◄

~~F~~→The first operation performed by the system is indexing. ~~In the indexing process collection of PDF files is provided to *Parser*~~ (1). The *Parser* ~~parses~~→converts the PDF files into text and extract meta H. ►*what meta information*◄ 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

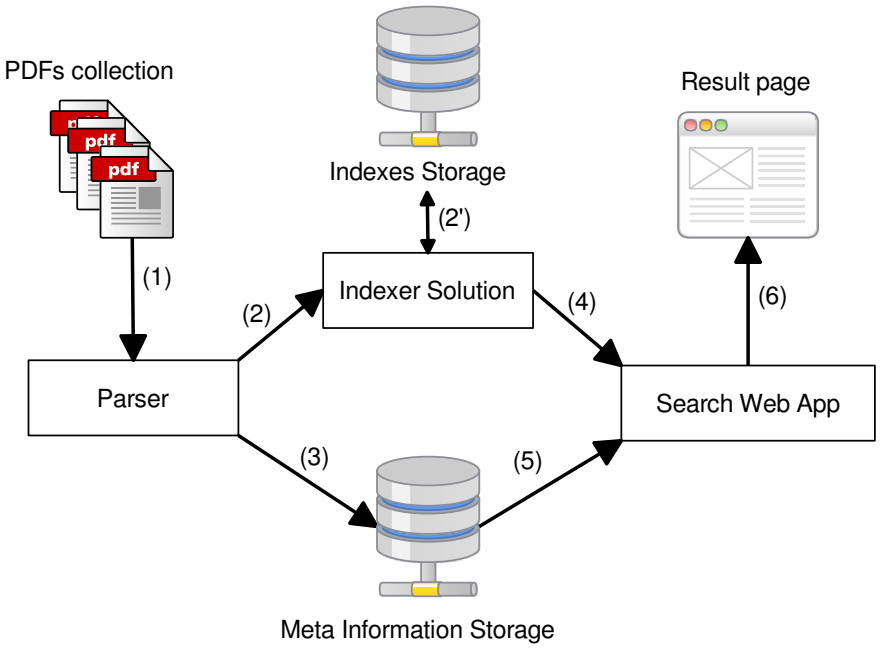

Figure 3.1: ~~Search Engine~~→change this to the name of the tool ;) high level architecture H. ►*The architecture diagram is not clear because it is a mix between an activity diagrams and a component diagram. There should be one diagram for each.*◄

representation of PDF files that doesn't require indexing is stored in *Meta Information Storage* (3). H. ►*the whole paragraph is not clear. Try to simplify it by just enumerating the steps in an another activity diagram where you explicitly specify the input and output if each step.*◄

In the querying process user searches for some phrase using *Search Web App* user interface. The phrase is analyzed by the *Indexer Solution*. *Indexer Solution* finds documents H. ►*what documents?? Remember that your thesis should be self-explanatory. I know that the term "documents" is used when dealing with Solr, but the reader doesn't. Try to be more clear.*◄ matching the query phrase in *Indexes Storage* (2') and sends them to *Search Web App* (4). If the user is interested in getting details regarding ~~the~~→a specific reference, the information H. ►*what information?*◄ will be looked up in the *Meta Information Storage* (5). Finally, the result is shown to the user as ~~a~~ an ~~html~~→HTML page (6).

The ~~N~~→next sections of ~~the~~→this chapter describe the implementation of each component in detail and show the reasons ~~of~~→behind choosing a particular solution.

3.2 Parser

It is practical to divide the work of the *Parser* into two phases: *Processing* and *Publishing*, as ~~on~~→in Figure 3.2. The output of the *Processing* phase is the input to the *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.~~

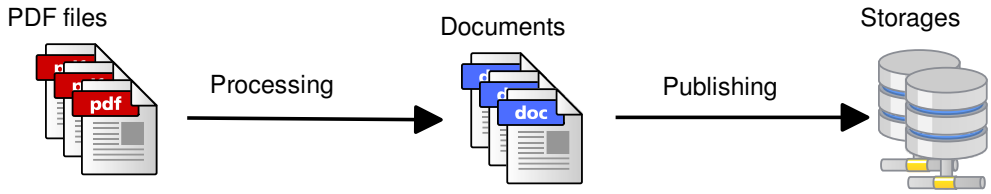

Figure 3.2: Parser workflow

3.2.1 Processing

H. ►*give this phase a better title and use it accordingly in the text.*◄ 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 p~~→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~~→from articles created recently. Thereby, building a universal parser is ~~practically impossible~~→very hard in practice. In our case, we try to identify common patterns covering the majority of the scientific article formats or at least the formats found in our collection of articles.

Processing phase starts with recursively walking though the directory tree of the collection library. While walking ~~though~~→through the directory, the *Parser* filters non-pdf files and parses and processes each PDF-file separately. We use~~d~~ Apache PDFBox library [1] to parse PDF-files into text. The library extracts full text from PDF-files, 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~~→. We designed an algorithm to break the PDF-text into sections.

Generally, we are interested in identifying a body of the document where we can find citations and reference~~s~~ blocks 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 sections. Based on

[1] Apache PDFBox, https://pdfbox.apache.org/

those keywords, one can extract different sections of a document. Figure 3.3 shows a sample text of a parsed pdf-document with keywords.

One can notice the following characteristics of scientific articles:

- The Body of a document comes before the references section.
- The Appendix or Author's biography sections can come after the references section.
- Each document contains ~~an~~→the "Abstract" and the "References" words and might contain the "Appendix" word. We call these words keywords.

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

body	references	appendix
Abstract ABSTRACT	References References: REFERENCES	Appendix APPENDIX

Table 3.1: Keywords identifying different sections in a document

After breaking ~~down a document~~→a document down into sections as shown ~~on~~→in Figure 3.3, the text ~~in sections are~~→is presented in one-column format. There are two aspects regarding this format. First, sentences can be ~~splitted~~→split by new line symbols ~~in~~→at the end of a line. Second~~ly~~, words can be ~~splitted~~→split by dash symbol ~~in~~→at the end of a line. ~~It was~~→We 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~~→continuous text.

As a result of the normalization step, we have a document divided into body and references sections. Before searching citations in the body of a document ~~it is reasonable to~~ →, we break the 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 be encountered inside ~~a~~ →the sentence itself, like an item of a numbered list or a name of a scientist. Besides, not all sentences end ~~by~~→with a period, ~~for example,~~→like the title of a section or an item of a list. ~~It was~~→We used Stanford CoreNLP library that employs natural language and machine learning techniques to extract sentences [9].

Next, we search for the citations in ~~a~~→the body ~~of a document~~ and for the bibliography links in ~~a~~→the references section. When an author makes a citation ~~it~~→he puts a link to a bibliographic reference in the sentence. It is common to use square brackets ([and]) to make

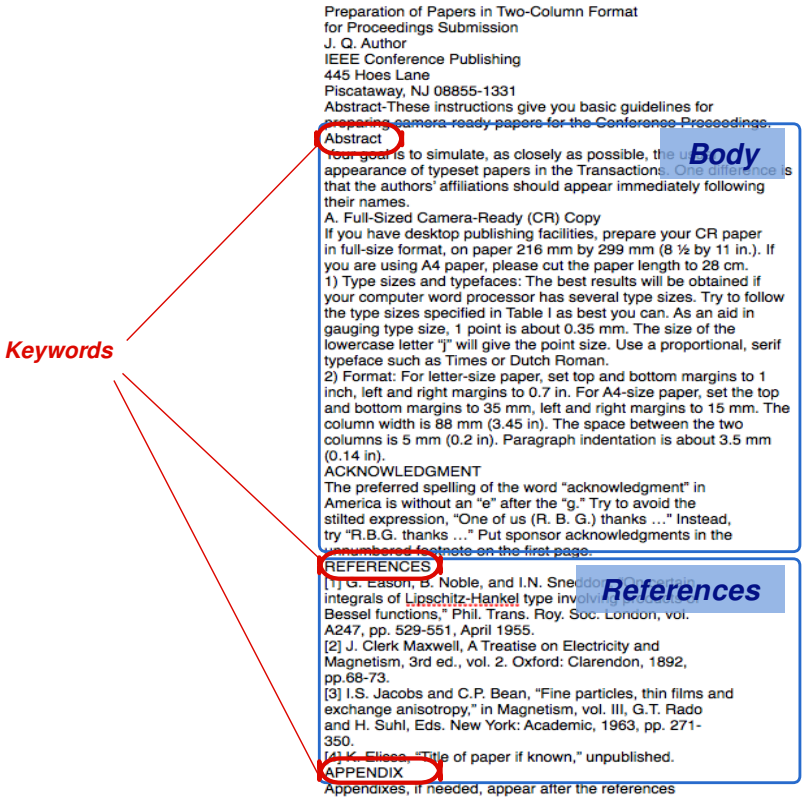

Figure 3.3: Sample text of the parsed scientific article. Keywords help to break the document into sections. H. ►*Why isn't this figure centred?*◄

a link to a bibliographic reference in the sentence. Thus, detecting square brackets in the text we can identify citations H. ►*convoluted. Try to write sentences in the normal order*◄. After analyzing some set of articles ~~it was revealed~~→, we found multiple ~~frequent~~ patterns in using square brackets for ~~citing~~→citations, as ~~showed~~→shown in Table 3.2.

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

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 list of regular expressions matching one of those patterns listing in Table 3.3. Extracting bibliographic links allow us to match citations with bibliographic links.

The pipeline 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. The objective point of extracting titles from bibliographic links is to collect citations referred to the the same source (scientific article). In general case, different formats of bibliographic links can identify the same source or scientific article. For example, an article may have different editions, published in different journals in different years or simply different authors may use different style formatting. What we consider to be identical for all bibliographic links citing the same paper is the paper's title.

Processing bibliographic links As usual we try to recognize common patterns belonging to majority of bibliographic links. Table 3.4 shows some examples of bibliographic links. First, it

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

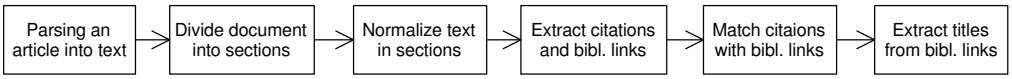

Figure 3.4: Pipeline of processing stage

was noticed that if bibliographic link contains some sort of quotes, for example, double quotes "" or single quotes '', then highly probably that a title is enclosed by these quotes. Then, some observations were made for bibliographic links without quotes. Very often, a bibliographic link structured as follows: it begins by listing paper's authors, then paper's title goes and then comes the rest of the link (see Figure 3.5). It turned out that Core NLP library used for breaking text into sentences is good in dividing a link into parts according to our view. In most of cases it's enough to take the second part of the bibliographic link to be a title.

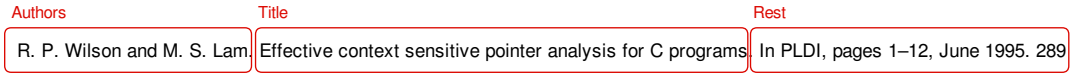

Figure 3.5: Structure of a bibliographic link

Conradi, R., Dyba, T., Sjoberg, D.I.K., and Ulsund, T., ”Lessons learned and recommendations from two large norwegian SPI programmes.” Lecture notes in computer science, 2003, pp. 32-45.”
P. Molin, L. Ohlsson, ‘Points & Deviations - A pattern language for fire alarm systems,’ to be published in Pattern Languages of Program Design 3, Addison-Wesley.
R. P. Wilson and M. S. Lam. Effective context sensitive pointer analysis for C programs. In PLDI, pages 112, June 1995. 289
Allen, Thomas B. Vanishing Wildlife of North America. Washington, D.C.: National Geographic Society, 1974.
I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In Proceedings of the 4th International Workshop on Mining Software Repositories, Minnesotta, USA, May 2007.

Table 3.4: Some examples of 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 reasonings in favor of using MongoDB additionally to Solr storage will be given below.

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 3.6 represents a structure of documents we publish to Solr.

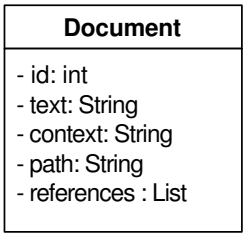

Figure 3.6: Document structure publishing to Solr

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 storage is very 'flat', which means that Solr doesn't allow to store even hierarchical data [25], [21]. 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.

Publishing to MongoDB MongoDB is a document-oriented NoSQL database that stores data in JSON-like documents with dynamic schema [2]. 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.

[2]MongoDB database, http://www.mongodb.org/

```
{
   "_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

3.2.3 Challenges

One of the challenge during parsing is finding cited sentences that do not contain any specific identifiers. Indeed the task is straightforward when a sentence contains square brackets, for example, '[34]' or '[Ali86]'. However, sometimes a link to bibliography can be composed only from the authors' names. In this case it becomes difficult to distinguish a citation from any other sentence.

Another challenge in using square brackets as citation identifiers arises when square brackets are used not as links to bibliography. For example, a parser might mix up an array in a code snippet with a link to bibliography. Usage of code snippets are common in computer science literature so it would be nice to have a method to distinguishing a snippet of source code from the natural text.

3.3 Indexator

As an Indexator we use Solr: solution from Apache Software Foundation built on Apache Lucene. Apache Lucene is an open source, information retrieval library that provides indexing and full text search capabilities [3]. While web search engines focus on searching content on the Web, Solr is designed to search content on corporate networks of any form. Some of the public service that use Solr as a server: Instagram (photo and video sharing social network), Netflix (movie hosting service), StubHub.com (public entertainment events ticket reseller).

Figure 3.7 illustrates a high level architecture of Solr. Solr is distributed as a Java web application that runs in any servlet container, for example, Tomcat or Jetty. It provides REST-like web services so external applications can make queries to Solr or index documents. Once the data is uploaded, it goes through text analysis pipeline. In this stage, different inspections to remove duplicates in the data or some document-level operations prior to indexing, or create multiple documents from a single one can be applied. Solr comes with variety of query parser implementations responsible for parsing the queries passed by the end search as a search string. For example, *TermQuery, BooleanQuery, PhraseQuery, PrefixQuery, RangeQuery, MultiTermQuery, FilteredQuery, SpanQuery* and others. Solr has xml configuration files (schema.xml and solrconfig.xml) to define the structure of the index and how fields will be represented and analyzed.

Next section describes configuration of Solr instance for our search system.

3.3.1 Solr Configuration

Solr installation requires JDK and any servlet container installed on a machine. Here configuration process for Apache Tomcat container is described. We need to download Solr distribution that can be found on official Solr home page [4]. Solr is distributed as an archive. After unzipping the archive, the extracts have following directories:

- **contrib/** - directory containing additional libraries to Solr, such as Data Import Handler, MapReduce, Apache UIMA, Velocity Template, and so on.
- **dist/** - directory providing distributions of Solr and other useful libraries such as SolrJ, UIMA, MapReduce, and so on.
- **docs/** - directory with documentation for Solr.
- **example/** - Jetty based web application that can be used directly.

[3] Apache Lucene, http://lucene.apache.org/core/
[4] Apache Solr, http://lucene.apache.org/solr/

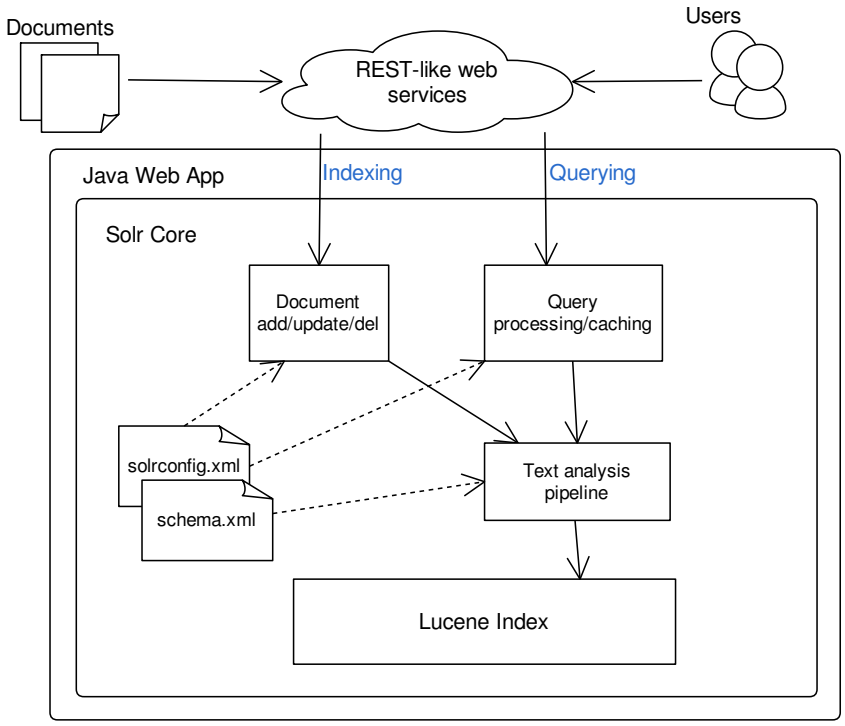

Figure 3.7: High level architecture of Solr system

- **Licenses/** - directory containing all the licenses of the underlying libraries used by Solr.

Copy the dist/solr.war file from the unzipped folder to $CATALINA_HOME/webapps/solr.war. Then point out to Solr location of home directory describing a collection. By collection in Apache Solr one imply a collection of Solr documents that represent one complete index. It's possible to choose one of the options:

- **Java options** one can use following command so that the container picks up Solr collection information from the appropriate location:

```
$export JAVA_OPTS="$JAVA_OPTS -Dsolr.solr.home=/opt/solr/example"
```

- **JNDI lookup** Or it is possible to configure the JNDI lookup for the java:comp/env/solr/home resource by pointing it to the Solr home directory. This can be done by creating a context XML file with some name (context.xml) in $CATALINA_HOME/conf/Catalina/localhost/context.xml and adding the following entries:

```
<?xml version="1.0" encoding="utf-8"?>
<Context docBase="<solr-home>/example/solr/solr.war" debug="0" crossContext="true">
<Environment name="solr/home" type="java.lang.String" value="<solr-home>/example/solr" ov
</Context>
```

The Solr home directory contains configuration files and index-related data. It should consist of three directories:

- **conf/** - directory containing configuration files, such as solrconfig.xml and schema.xml
- **data/** - default location for storing data related to index generated by Solr
- **lib/** - optional directory for additional libraries, used by Solr to resolve any plugins

Configuring Solr instance requires defining a Solr schema and configuring Solr parameters.

Defining Solr schema Solr schema is defined in schema.xml file placed to conf/ directory of Solr home directory. Solr distribution comes with a sample schema file that can be changed for the needs of the project. Schema file defines the structure of the index, including fields and field types. The basic overall structure of schema file is following:

```
<schema>
  <types>
  <fields>
```

```
    <uniqueKey>
    <copyField>
  </schema>
```

The basic unit of data in Solr is document. Each document in Solr consist of fields that described in schema.xml file. By describing data in schema.xml Solr understand the structure of the data and what actions should be performed to handle this data. Here is an example of a field in schema file:

```
<field name="id" type="integer" indexed="true" stored="true" required="true"/>
```

Table 3.5 lists and explains major attributes of field element.

Name	Description
Default	default value if it is not read while importing a document
Indexed	true if field should be indexed
Stored	when true a field is stored in index store and is accessible while displaying results
compressed	when true a field will be zipped, applicable for text-type fields
multiValued	if true, field can contain multiple values in the same document.

Table 3.5: Major attributes of field element in schema.xml

Here is a fragment of schema file defining fields of a document in Citation Search Engine collection:

```
 <fields>
  <field name="_version_" type="long" indexed="true" stored="true" multiValued="false"/>
  <field name="id" type="string" multiValued="false"/>
  <field name="text" type="text_en" indexed="true" multiValued="false"/>
  <field name="context" type="string" indexed="false" multiValued="false"/>
  <field name="path" type="string" indexed="false" multiValued="false"/>
  <field name="reference" type="string" indexed="false" stored="true" multiValued="true" />
 </fields>
```

Every document represent a citation with matching bibliographic references. In schema file we indicate that we want to index text field which is citation text. We store id of a citation, that is a generated value, calculated from the hash of the citation string. Specifying id is particularly useful for updating documents. We also store a context of a citation and path to the scientific article where citation was found. As citation can refer to multiple sources, we make reference field multivalued.

In schema configuration file one can define field type, for example, string, date or integer and map them to Java class. This can be handy to define custom type. A field type includes following

information:

- Name
- Implementation class name
- If field type is TextField, it will include a description of field analysis
- Field attributes

A sample field type description:

```
<fieldType name="text_ws" class="solr.TextField" positionIncrementGap="100">
   <analyzer>
   <tokenizer class="solr.WhitespaceTokenizerFactory"/>
   </analyzer>
</fieldType>
```

Other elements in the Solr schema file listed in Table 3.6:

Name	Description
UniqueKey	specifies which field in documents is a unique identifier of a document, should be used if you ever update a document in the index
copyField	used to copy field's value from one field to another

Table 3.6: Description of some elements in schema.xml

Once the schema is configured next step is to configure instance itself.

Configuring Solr parameters To configure Solr instance itself we need to describe solrconfig.xml and solr.xml files.

solr.xml The solr.xml configuration is located in solr home directory and used for configuration of logging and advanced options to run Solr in a cloud mode.

solrconfig.xml The solrconfig.xml configuration file primarily provides you with an access to index-management settings, RequestHandlers, listeners, and request dispatchers. The file has a number of complex sections and mainly is changed when a specific need encounter.

3.3.2 Enhanced search features

Solr provides a number of additional features that can enhance search system. One of the feature we use is synonyms. To use this feature you need to specify synonyms.txt file with listed

synonyms. This file is used by synonym filter to replace words with their synonyms. For example, a search for ”DVD”may expand to ”DVD”, ”DVDs”, ”Digital Versatile Disk” depending on the mapping in this file. This file can be also used for spelling corrections. Here is an example of synonyms.txt file:

```
GB, gib, gigabyte, gigabytes
MB, mib, megabyte, megabytes
Television, Televisions, TV, TVs
Incident_error, error
```

Additionally, there are other configuration files that appear in the configuration directory. We are listing them in Table 3.7 with the description of each configuration:

Name	Description
protwords.txt	file where you can specify protected words that you do not wish to get stemmed. So, for example, a stemmer might stem the word ”catfish” to ”cat” or ”fish”.
spellings.txt	file where you can provide spelling suggestions to the end user.
elevate.txt	file where you can change the search results by making your own results among the top-ranked results. This overrides standard ranking scheme, taking into account elevations from this file.
stopwords.txt	Stopwords are those that will not be indexed and used by Solr in the applications. This is particularly helpful when you really wish to get rid of certain words. For example, in the string, ”Jamie and joseph,” the word ”and” can be marked as a stopword.

Table 3.7: Additional configuration files in Solr

3.4 Meta Information storage

As a meta information storage we used MongoDB. MongoDB is a NoSQL document oriented storage written in C++. Data in MongoDB is stored in JSON-like documents with a dynamic schema. The format of stored data is called BSON, which stands for Binary JSON. BSON is an open standard developed for human readable data exchange [5]. MongoDB requires a little amount of configuration to start to work with.

[5]BSON specification, http://bsonspec.org/

3.4.1 Configuration

Once MonogDB distributive is downloaded is very easy to set up a database server. All we need to start the MongoDB server is to type *mongod* command. In our case we would like to specify database location with –dbpath parameter and default listening port:

```
> mongod --dbpath /home/aliya/mongodb2 --port 27272
```

MongoDB provides REST API, to enable REST API use parameter –rest:

```
> mongod --dbpath /home/aliya/mongodb2 --port 27272 --rest true
```

The simple way to communicate with MongoDB server is to use mongo shell, in our case we specify –port parameter to connect to our instance of MongoDB:

```
> mongo --port 27272
```

Compared to relational databases MongoDB operates with terms collection, which is equivalent to table, and document, which is equivalent to record in relational databases. MongoDB doesn't require creating databases and collections explicitly. Databases and collections can be created while starting to use MongoDB. To see list of databases or collections, type show dbs in mongo shell:

```
> show dbs
```

Mongo shell allows to make queries, updates, deletes on collections, get various statistics on data and server usage, and manipulate with data with map-reduce interface, full documentation can be found on the official web site [6].

3.5 Web search interface

Web Search interface is a Java web application running in a servlet container. Figure 3.8 shows an architecture of a web search application. The application is based on MVC (model-view-controller) architectural pattern implemented with Struts framework [7]. The application communicates with Solr via Solr REST API and with Mongo database via Java database connector.

3.5.1 Citation search page

3.5.2 Details page

[6] MongoDB database, http://www.mongodb.org/
[7] Struts framework, https://struts.apache.org/

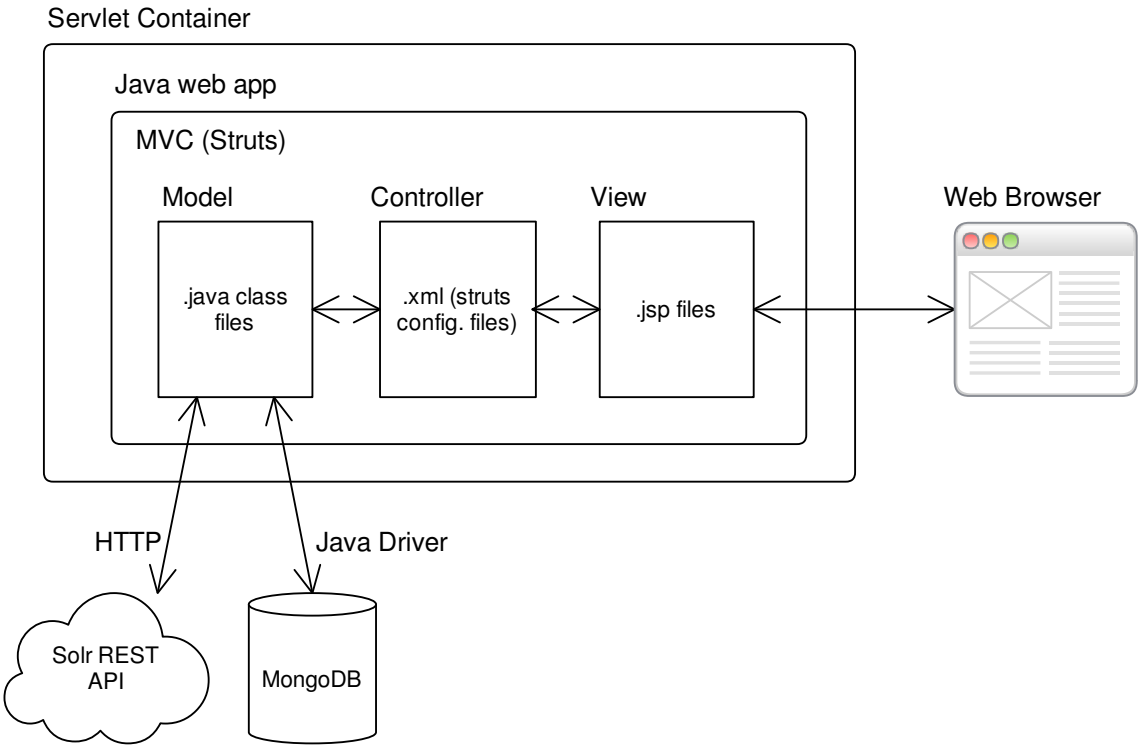

Figure 3.8: Architecture overview of web search application

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] Marc Bertin and Iana Atanassova. Semantic enrichment of scientific publications and metadata : Citation analysis through contextual and cognitive analysis. *D-Lib Magazine*, 18:8, 2012.

[2] Marc Bertin and Iana Atanassova. Extraction and characterization of citations in scientific papers. In Valentina Presutti, Milan Stankovic, Erik Cambria, Ivn Cantador, Angelo Di Iorio, Tommaso Di Noia, Christoph Lange, Diego Reforgiato Recupero, and Anna Tordai, editors, *Semantic Web Evaluation Challenge*, volume 475 of *Communications in Computer and Information Science*, pages 120–126. Springer International Publishing, 2014.

[3] Shannon Bradshaw. Reference directed indexing: Redeeming relevance for subject search in citation indexes. In Traugott Koch and IngeborgTorvik Slvberg, editors, *Research and Advanced Technology for Digital Libraries*, volume 2769 of *Lecture Notes in Computer Science*, pages 499–510. Springer Berlin Heidelberg, 2003.

[4] Shannon Glenn Bradshaw. *Reference directed indexing: Indexing scientific literature in the context of its use*. Northwestern University, 2002.

[5] 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.

[6] Gaizka Garechana, Rosa Rio, Ernesto Cilleruelo, and Javier Gavilanes. Visualizing the scientific landscape using maps of science. In Suresh P. Sethi, Marija Bogataj, and Lorenzo Ros-McDonnell, editors, *Industrial Engineering: Innovative Networks*, pages 103–112. Springer London, 2012.

[7] Eugene Garfield et al. Science citation index-a new dimension in indexing. *Science*, 144(3619):649–654, 1964.

[8] 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.

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

[10] M. M. Kessler. Bibliographic coupling between scientific papers. *American Documentation*, 14(1):10–25, 1963.

[11] Richard Klavans and Kevin W. Boyack. Toward a consensus map of science. *J. Am. Soc. Inf. Sci. Technol.*, 60(3):455–476, March 2009.

[12] Ray R Larson. Bibliometrics of the world wide web: An exploratory analysis of the intellectual structure of cyberspace. In *PROCEEDINGS OF THE ANNUAL MEETING-AMERICAN SOCIETY FOR INFORMATION SCIENCE*, volume 33, pages 71–78, 1996.

[13] Loet Leydesdorff, Stephen Carley, and Ismael Rafols. Global maps of science based on the new web-of-science categories. *CoRR*, abs/1202.1914, 2012.

[14] Bertin M, Descls J. P, Djioua B, Krushkov Y, and Lalicc Umr. Automatic annotation in text for bibliometrics use.

[15] Qiaozhu Mei and ChengXiang Zhai. Generating impact-based summaries for scientific literature. In *Proceedings of ACL-08: HLT*, pages 816–824, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[16] Preslav I. Nakov, Ariel S. Schwartz, and Marti A. Hearst. Citances: Citation sentences for semantic analysis of bioscience text. In *In Proceedings of the SIGIR04 workshop on Search and Discovery in Bioinformatics*, 2004.

[17] Hidetsugu Nanba and Manabu Okumura. Towards multi-paper summarization reference information. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 926–931, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[18] 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.

[19] 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.

[20] Vahed Qazvinian and Dragomir R. Radev. Scientific paper summarization using citation summary networks. In *Proceedings of the 22Nd International Conference on Computational Linguistics - Volume 1*, COLING '08, pages 689–696, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[21] Jonathan Rochkind. Thinking like solr its not an rdbms. https://bibwild.wordpress.com/2011/01/24/thinking-like-solr-its-not-an-rdbms/.

[22] Henry Small. Co-citation in the scientific literature: A new measure of the relationship between two documents. *Journal of the American Society for Information Science*, 24(4):265–269, 1973.

[23] Henry Small. Visualizing science by citation mapping. *J. Am. Soc. Inf. Sci.*, 50(9):799–813, July 1999.

[24] Linda C. Smith. Citation analysis. `https://www.ideals.illinois.edu/bitstream/handle/2142/7190/librarytrendsv30i1i_%20opt.pdf?sequence=1`, 1981.

[25] Solr Wiki. Why use solr? http://wiki.apache.org/solr/WhyUseSolr.

A

User guide for Citation Search Engine deployment

A.1 Solr Installation

A.2 MongoDB Installation

A.3 Running parser

A.4 Search interface deployment