Building Multilingual Search Index using open source framework

Arjun Atreya V[1] Swapnil Chaudhari[1] Pushpak Bhattacharyya[1] Ganesh Ramakrishnan[1]

(1) Deptartment of CSE, IIT Bombay
{arjun, swapnil, pb, ganesh}@cse.iitb.ac.in

ABSTRACT

This paper presents a comparison of open source search engine development frameworks in the context of their malleability for constructing multilingual search index. The comparison study reveals that none of these frameworks are designed for this task. This paper elicits the challenges involved in building a multilingual index. We also discuss policy decisions and the implementation changes made to an open source framework for building such an index. As a main contribution of this work, we propose an architecture that can be used for building multilingual index. It also lists some of the open research challenges involved.

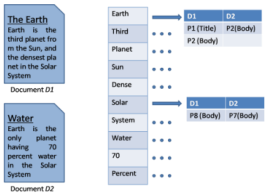
KEYWORDS: Inverted Index, Multilingual Index, Search Engine Framework

Proceedings of the 3rd Workshop on South and Southeast Asian Natural Language Processing (SANLP), pages 201–210,
COLING 2012, Mumbai, December 2012.

1. Introduction

There are lots of open source frameworks available to build a search engine. These open source frameworks provide multiple features to build an inverted index of web documents used for information retrieval. Some features like Scalability, term storage, document posting list storage *etc*, are common across these frameworks. These frameworks facilitate customization of building index to make it compatible for the desired application.

To retain the structure of a document in an inverted index, *field based* indexing is used. Instead of viewing a document as a collection of terms, the document is viewed as a collection of fields and the field as a collection of terms. Each document that needs to be indexed is parsed and terms in the document are grouped into fields prior to indexing. The conceptual view of field based inverted index is shown in the figure 1. Figure 1(a) shows two documents as is. Figure 1(b) shows a view of inverted index built for these documents.



1(a) 1(b)

FIGURE 1 - Conceptual view of an inverted index

All terms in the document are indexed, and a document posting list is created for each indexed term. For every document containing an indexed term, there is a posting list. This posting list contains position information of the indexed term and the field in which this indexed term is present in the document. Field information helps in prioritizing the document for a given search.

2 Overview of open source search frameworks

There are lots of works on building the inverted index using an open source framework. The most popular indexing library is Apache Lucene (Apache Lucene, 2011). Lucene is not a complete search engine framework, but an indexing library used to generate inverted index from crawled documents. Lucene needs to be plugged in with a crawler in order to index web documents. Apache Lucene provides facilities for customizing the library and makes it easily pluggable with the crawler that is being used.

Apache Solr (Apache Solr, 2012) is an enterprise indexing solution built on top of Lucene. Along with indexing, Solr provides features to add, delete and modify documents in an index. Solr also provides basic search facilities that include faceted search, highlighting *etc*. One of the advantages of Solr is to seamlessly add documents to the index, hence reducing the down time of the application.

Apache Nutch (Apache Nutch, 2005) is an open source crawler is built using Java. This project was initiated as a part of Apache Lucene project. Nutch is a scalable crawler framework used for crawling web documents. Heritrix (Heritrix, 2012) is also a web based crawler built using Java. Heritrix provides almost all features of Nutch along with good scalability. The comparison study between the two suggests that Heritrix is better for Multimedia retrieval, whereas Nutch with Hadoop (Apache Hadoop, 2012) is best suited for distributed text retrieval.

Two most popular search frameworks used in research are Terrier (Terrier, 2011) and Lemur (Lemur, 2012). Both of these projects do not include crawler and are not designed for web based search. However, these frameworks are highly scalable and best suited for crawling local file system, TREC Collection, CLEF collection, FIRE collection *etc.*

Other open source search engines built on top of Lucene include Compass (Compass, 2010), Oxyus (Oxyus, 2010), Lius (Lius, 2010), Regain (Regain, 2004). All these search engines have similar features with different capabilities. Compass and Oxyus are designed for crawling web documents, whereas Regain is designed for local system crawl. YaCy (YaCy, 2007) is a decentralized web search framework. This project deals with the usage of networked systems to store a shared index. YaCy provides consistent retrieval for all users without censoring data from the shared index. A small scale search engine Swish-e (Swish-e, 2007), is designed for crawling web pages but for a scale of less than a million documents. MG4J (MG4J, 2005) is a scalable full text search engine built using Java. This framework uses quasi-succinct index for searching. It supports large document collection and indexes large TREC collection without much effort. Google also provides an open source search engine framework called Hounder (Hounder, 2010). Hounder is built on top of Lucene for web retrieval.

Many of the open source search engine frameworks mentioned above are compared for their efficiency in (M. Khabsa *et. al.*, 2012). Each of these frameworks provides certain good qualities. Comparison study in (M. Khabsa *et. al.*, 2012) shows that Lucene takes more time for indexing but storage size of the index is minimized. Similarly MG4J takes lesser time to index. However, these comparison studies are done for certain parameters. There is no such comparison study that considers all parameters of a web crawler.

Keeping in mind the humongous growth of the web and also the need for common solution to search multiple language documents in the web, it is important to evaluate parameters pertaining to multilingual documents while building index. These parameters include language tagging of the document in an index, retrieval efficiency for a specific language and so on.

These frameworks are designed for generating monolingual index. None of the before mentioned frameworks provides all features for generating multilingual index. Frameworks demand changes in the architecture to handle language information of the document while indexing them. Changes in the architecture for one of these open source frameworks called Solr is proposed in this work.

This work details the importance of these parameters. Section 3 describes features of a web crawler that are required to build an index for text retrieval. Features discussed in

this section indicate development of web crawlers in the recent past. Our proposed architecture to build multilingual index is introduced in section 4. Section 5 lists several research problems to be addressed in the area of building multilingual index.

3. Features of Web based Crawler/Indexer

A crawler should possess certain features to build a web text retrieval engine. The crawler must be able to process different document formats like HTML, XML, doc, pdf, rtf, *etc*. These documents are fetched from the Internet and parsed individually to get the clean content. With the growth of commercial online advertisement industry, cleaning the HTML content is becoming more challenging. Obtaining the clean text helps us in indexing documents efficiently (Apache Tika, 2012).

There are several components involved in the crawler, *viz*., fetcher, parser and indexer. Each of these components is customized to enhance the capabilities of a search engine. In this section we discuss several features of the web crawler that are necessary for a good web search engine. The challenges involved in building multilingual index with respect to each of these features are discussed.

3.1 Incremental crawl/index

Crawling is a continuous process of fetching new web documents from the Internet. This is because, all search engines try to achieve a near real time search by making the latest information available to the user . To achieve this, it is important to seamlessly add documents to the index as and when they are fetched. This process of seamlessly adding documents to an existing index is called incremental crawling.

A process need to be in place that adds latest crawled documents to the index while crawling of other pages is still going on. A crawler process that is continuously running and seamlessly adding documents to the index is essential for a near real time search. The open source framework chosen for this task should possess these qualities. There should not be considerable downtime of the index while updating. This downtime would restrict the user from searching documents in the web.

Another aspect of the incremental crawl is re-crawling of already fetched documents. Re-crawling involves identifying changes in fetched documents and indexing them again. This involves deleting a previous version of the document and adding new one to the index.

3.2 RSS Feeds

There are various categories of data in the web that need to be crawled. One such categorization is with respect to the format of a document like HTML, XML, pdf, doc, *etc*. Another categorization is with respect to the modification frequency of the website. There are several websites that remain unchanged for a longer period of time like tourism information sites. Whereas, there are several other sites that change very frequently and these changes need to be tracked in order to crawl them continuously.

As discussed earlier, all search engines try to be real-time or near real-time search engines. To achieve this, crawler should provide a facility to keep track of changes happening in a website. In recent times many websites are being developed as search

engine friendly and maintain a record of all changes happening in that site. Websites log all changes in the form of RSS feeds at a single location. This helps the crawler to monitor a particular or small set of RSS feed documents to know if there are any changes in that site.

RSS feeds are generally in XML format containing links to changes done in that site along with metadata information. Crawler need to handle these pages in a special way *i.e.*, after fetching RSS feeds, crawler need to parse the page for changes. If changes exist in the RSS feed page then all out links in the RSS feed document are fetched and indexed. Currently there are many open source crawlers like Nutch which support this feature but older versions of these crawlers do not support crawling of RSS feeds.

3.3 Web graph

Crawling of web pages is driven by links present in the document. While parsing the fetched document all out-links of the document are recorded for crawling in the next depth. This process continues till the specified depth is reached. Hence link information in all documents is a vital part of the information for crawling. This link information is processed to build a web graph of crawled pages. The web graph thus built can then be used for scoring the document or for ranking retrieved results.

Several algorithms use the web graph generated while crawling. These algorithms include the following.

- Recognizing hub pages: This process involves categorizing the page as a hub page (main page or home page of a site). The number of in-links and out-links of a page is used to do this task.
- Ranking algorithm: There are several ranking algorithms designed based on the link graph among crawled pages. Google's Page Rank algorithm (Arasu *et. al.*, 2002) is one of the famous ranking principles built based on the web graph of crawled pages.
- Document scoring: While indexing, each document is given certain boosting so as to indicate the importance of a document over other crawled documents. The intuitive methodology is to use the ratio of in-link to out-link score to identify the boosting factor of the document.

Apart from the above mentioned applications, web graphs are also used to find the relation between crawled documents, clustering of related documents in the crawl, *etc.*

4. Our proposed architecture

Based on the popularity of open source search frameworks and their method of implementation, we chose tools for building a multilingual search index. The latest version of Nutch with Hadoop was chosen as a crawler and Apache Solr for building the index of crawled documents. Both Nutch and Solr support customization of modules to make them adaptable to a desired application. The desired application for which these tools are used is to develop monolingual web search engine for 9 Indian languages, *viz.*, *Assamese, Bengali, Gujarati, Hindi, Marathi, Odiya, Tamil* and *Telugu*.

During the development process, several implementation challenges were encountered. Some of these challenges were handled by developing new modules like language

identifier. These were additional functionality requirements. A few other problems were addressed by building resources for modules to handle different language phenomenon. The named entity list and multiword list for each of these 9 Indian languages are examples of building resources. But there exists some implementation challenges that require a policy decision or an architectural change. In this section we discuss these policy decisions and architectural changes of Solr to make it suitable for the desired application.

Before discussing changes in the architecture of a system, it is important to detail the infrastructure that is being used for the application. Changes in policy decisions should also consider the infrastructure of a search system. The crawling process is distributed across 12 systems and the generated single multilingual index is stored in a high end search server that hosts web application.

The experience of crawling around 6 million documents possesses a few implementation challenges. One of them being re-crawling of documents already crawled. Building a mechanism to re-crawl documents that are already crawled require some insight into types of documents that are being crawled. There were documents from different domains like news sites, blogs, general sites and encyclopedia sites. We observed that frequency of updation in these sites vary enormously. News sites update on hourly basis, while blogs gets updated at a frequency of few days. General web sites change more slowly and encyclopedia sites hardly change at all. These differences make it impossible to have same re-crawl period for all documents in the crawl.

A policy decision was made to have different re-crawl periods for different kinds of sites. Nutch framework provides a facility for adaptive fetching. This method of fetching monitors a document to be crawled and adapts the re-crawl period based on whether the document has changed between two successive crawls. This algorithm reduces/increases the re-crawl period of the document by a fixed interval if the document has changed/not changed respectively from the previous crawl. Policy decisions involve bootstrapping the re-crawl period and also to decide upon the interval for different types of documents.

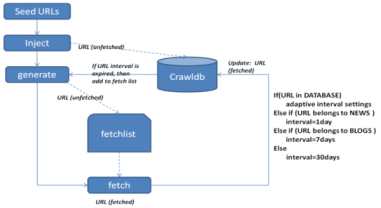


FIGURE 2 - Architecture depicting the re-crawl periods of a document

Figure 2 depicts decisions taken on re-crawl periods for different types of documents to be crawled. The bootstrapping values are used for initial fetching and then adaptive fetching is run for each of these documents. The challenge of categorizing documents is done at the level of domain name. List of sites belonging to news and blogs are built. These lists are used to fix the bootstrapping re-crawl period for a crawled document. Also, this list is prepared separately for each language.

Another important issue involves building multilingual index using Solr. Solr doesnot support use of single multilingual index, hence, architecture of Solr need to be modified. In Solr, all language analysis filters are listed in a configuration file called *schema.xml*. This file lists all filters to be called over a token stream of the document in an order. Token stream is a list of tokens that need to be indexed for a particular document. Architecture of the Solr system applies all the filters listed in *schema.xml* on the given token stream.

In the context of multilingual index, the language analyzer corresponding to the language of a document should be invoked while indexing. This demands a change in the architecture of Solr so that along with the token stream object, a language tag corresponding to the document is also passed in each module's API. Further the *schema.xml* file was changed to accommodate the language tag with each of these filters. Each filter listed in the configuration file has an associated language tag for which the filter is applicable. Figure 3 shows the differences in the *schema.xml* file before and after the architectural change in Solr. The filters listed in the *schema.xml* after changes indicate that only those filters whose language tag matches with the document's language tag are invoked while indexing.

FIGURE 3 - Snapshot of *schema.xml* (left) without change (right) with change

It is important to note that none of the open source frameworks discussed in the initial sections of this paper deal with building a multilingual index. To build a multilingual index, change in the architecture is must. This is irrespective of the framework chosen. In this direction, change in the architecture of open source indexing framework like Solr is an important step.

5. Research challenges in building multilingual index

The ranking algorithm drives the issues listed in this section. This is because, the impact of these issues is studied with respect to the ranking of retrieved documents. Most of the ranking algorithms used in text retrieval systems are based on variations of the tf (term frequency)-idf (inverse document frequency) based scoring. Tf–idf values are calculated based on the number of documents and terms in the document while building index.

In case of the multilingual index, it is possible to have the same indexed term that appears in multiple languages. For example, the term "दुर्गा" *{Durgaa; name of Indian Goddess}* is common for languages like *Marathi, Hindi, Konkani* and *Nepali*. This is because prior mentioned languages share the same script. This phenomenon will have

an impact in evaluating idf value of the term. During creation of the multilingual index, every term is indexed with a list of documents in which the term is present. The list includes documents from all languages for the term that is same across languages. If the goal of a retrieval system is to retrieve documents for a query irrespective of languages then this phenomenon will not have any negative impact but, in case of cross lingual search or monolingual search, this kind of index would mislead. The idf value captured by this term is inclusive of all language documents, this is not desired for monolingual search of a particular language.

The intuitive solution to this problem is to tag the language for each document that has been indexed. Even though this methodology helps in identifying the language of a document while retrieving, this would not be a complete solution because of misleading idf score of the term. The idf scores are not altered during index building as only language tagging is done to the document posting list. Conceptually a term and its corresponding posting list should be present in the index as many times as that of the number of languages in which the term is present.

One of the solutions to this problem involves having language based multiple indices. This however solves the problem of recognizing a document's language as well as having precise idf score. On the other hand, this solution demands a change in the infrastructure and brings in performance issues. Variation in the number of documents of each language is so large in the web that few languages dominate the web content hence language based indexing is not prescribed. Another solution to this problem is to change the structure of the conventional inverted index and force the index to store multiple entries of the same term for each language in which it is present. This still stands as an open issue in building a single multilingual index.

Conclusion and perspectives

This paper dealt with building a multilingual index using an open source framework. We proposed an architecture by customizing an open source framework called Solr to build a multilingual index. Policy decisions taken and implementation challenges faced in this process are explained in detail. A wide range of open source search engine frameworks were discussed with their benefits and limitations. A comparative study based on several parameters of these search engines were done to get better perspective with respect to building an index.

Several features of the crawler were detailed for web based text retrieval system. These features were discussed so as to indicate the importance of the continuous crawling process and seamless modifications that need to be done to the index. The proposed architecture is already been implemented and the evaluation process of the search engine is in progress.

Few research challenges involved in building multilingual index were introduced. Having an indexed term sharing the script in multiple languages poses a challenge of getting wrong document frequency. This leads to an idea of having separate index for every language and also demands a change in infrastructure. These issues are yet to be addressed to build an efficient multilingual index.

Acknowledgments

Thanks to AAI Group, CDAC Pune for providing the crawling infrastructure for this application.

References

Apache Lucene (2011) *http://lucene.apache.org/*

Apache Solr (2011) *http://lucene.apache.org/solr/*

Apache Nutch (2005) *http://nutch.apache.org/*

Heritrix (2012) *https://webarchive.jira.com/wiki/display/Heritrix/Heritrix*

Apache Hadoop (2012) *http://hadoop.apache.org/*

Terrier (2011) *http://terrier.org/*

Lemur (2012) *http://www.lemurproject.org/*

Compass (2010) *http://compass-project.org/*

Oxyus (2010) *http://sourceforge.net/projects/oxyus/*

Lius (2010) *http://sourceforge.net/projects/lius/*

Regain (2004) *http://regain.sourceforge.net/*

YaCy (2007) *http://yacy.de/*

Swish-e (2007) *http://swish-e.org/*

MG4J (2005) *http://mg4j.di.unimi.it/*

Hounder (2010) *http://code.google.com/p/hounder/*

M. Khabsa, Stephen Carman, S. R. Choudhury and C. L. Giles (2012)A Framework for Bridging the Gap Between Open Source Search Tools, *In proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*

Arasu, A. and Novak, J. and Tomkins, A. and Tomlin, J. (2002). "PageRank computation and the structure of the web: Experiments and algorithms". *Proceedings of the Eleventh International World Wide Web Conference*, *Poster Track*. Brisbane, Australia. pp. 107–117.

Apache Tika - The Parser interface (2012) *https://tika.apache.org/1.2/parser.html*