Search Engine Using Java

Project Report

Guided By
Dr. Luis Rueds
Mr. Abed Alkhateeb

University of Windsor
School of Computer Science

Table of Contents

1. Introduction

As part of our term project, a search engine will be developed which will take search string and outputs the number of occurrence-sorted list of web pages in which particular key-word can be found. What our search engine does is it takes the input from the user which is string, then it processes the query and returns the results which are the locations of the files where the keyword was found. Moreover, the results are sorted on basis of their page score. As this is a static search engine, we are using the 100 files provided by Dr Luis Rueda. But, more files can be added in no time if required.

1.1 Technologies Used

Languages: Java, JSP, HTML (Under Development)
Platform: Eclipse Java EE
Libraries: Apache Lucene Core 2.3.0

1.2 Minimum Hardware Requirements

RAM: 256 MB minimum
HDD: 20 MB free space

1.3 System Architecture

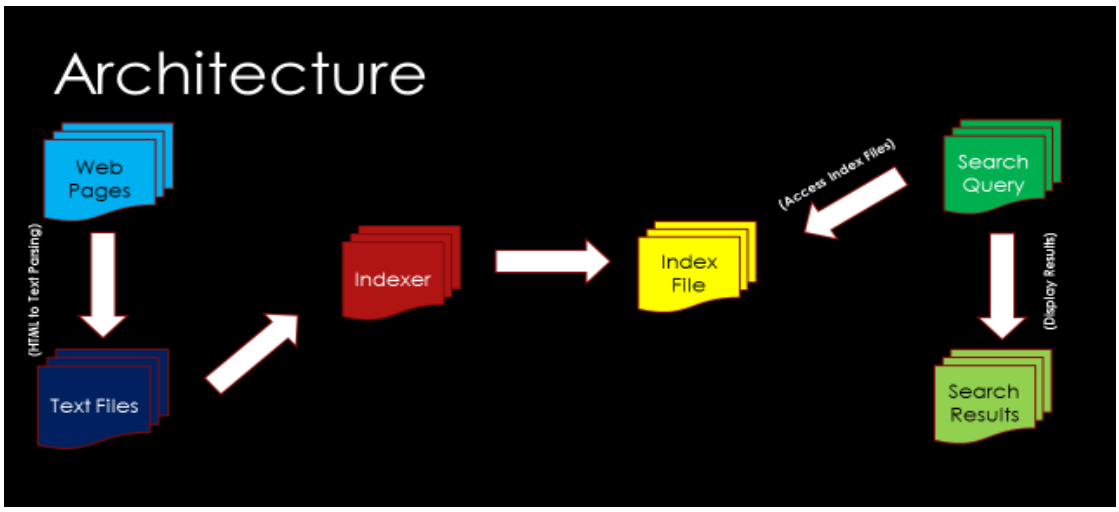


Fig. System Architecture

Above picture gives an idea about how the flow of the system will be. There will be few web pages which are first converted to text files. Now text files will be input to Indexer which will create and store index file in the directory. This index file will be accessed by the searcher whenever user enters the query. And then the results are displayed to the user.

2. Overall Description

2.1 Use Case Diagram

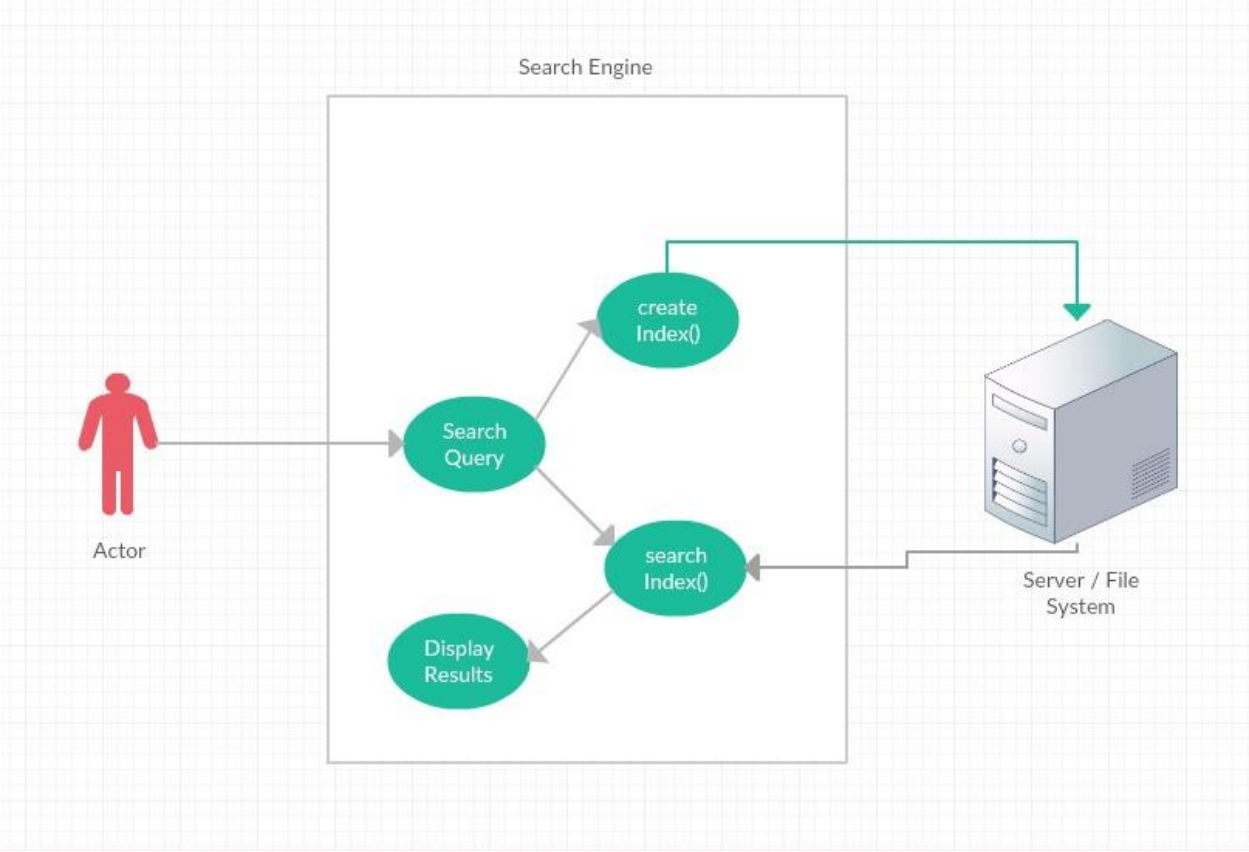


Fig. Use Case Diagram

The use case diagram explains the overall system. This is how all the operations will be carried out in our system. Note that I have not mentioned HTMLtoText conversion as it is a different system. So here we assume that we already have some text files stored in a directory. Now when user starts search engine, new index will be automatically created without letting the user know. This process is too fast. So now user will enter search query and the results will be displayed to user. It can be seen from the figure that the index file can be stored on server or currently it is stored in the file system.

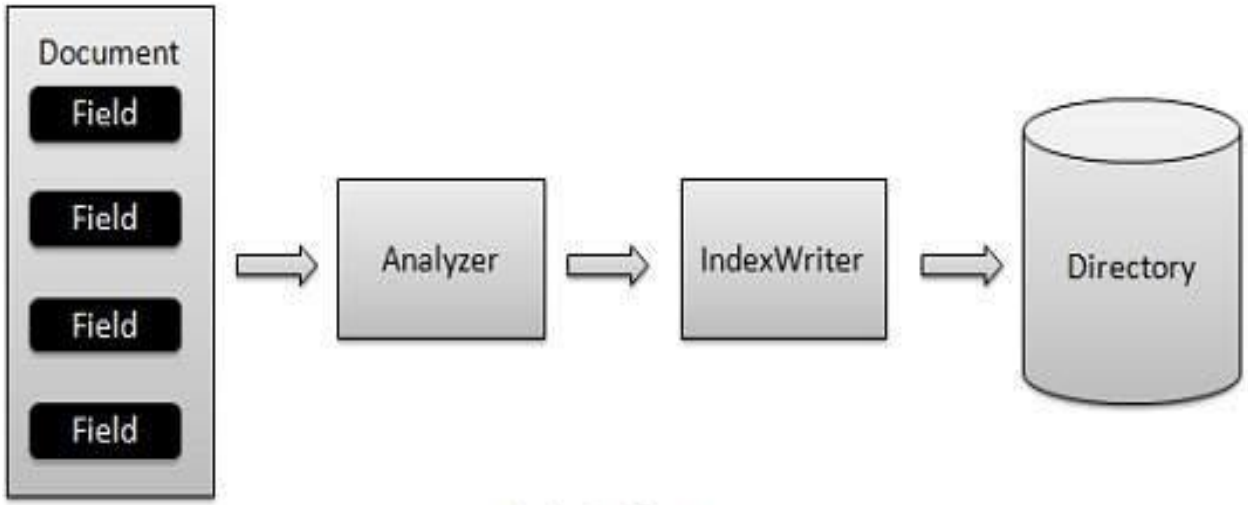
2.2 Features

1. Fast indexing: Indexing and searching of 100 webpages is done in maximum 5 seconds.
2. Scoring: Top results are returned first always.
3. Multi-word Search: It allows searching multi word queries
4. HTML to Text conversion

2.3 Implementation

Indexing

Apache Lucene core 2.3.0 is used for Indexing files. Files are the text files.



Indexing Process

Above figure shows how the documents will be indexed. As we can see, Lucene indexing uses data structures to store the index.
Following are the classes used for indexing the documents.

1.) Analyzer()This statement creates an instance of the StandardAnalyzer class, which is in charge of extracting tokens out of text to be indexed. StandardAnalyzer is just one implementation of the abstract class Analyzer; other implementations, such as SimpleAnalyzer, exist.

2.) IndexWriter() This statement creates an instance of the IndexWriter class, which is a key component in the indexing process. This class can create a new index or open an existing index and add documents to it. You might notice that its constructor accepts three parameters. The first parameter specifies the directory that stores the index files; the second parameter specifies the analyzer that will be used in the indexing process; the last parameter is a Boolean variable. If true, the class creates a new index; if false, it opens an existing index.

3.)

```
Document document = new Document();
document.add(new Field(FIELD_PATH, path,Field.Store.YES,
Field.Index.UN_TOKENIZED));
document.add(new Field(FIELD_CONTENTS, reader));
indexWriter.addDocument(document);
```

The first line creates an instance of the Document class, which consists of a collection of fields. You can think of this class as a virtual document, such as an HTML page, a PDF file, or a text file. The fields in a document are often the attributes of a virtual document. Take an HTML page, for example: Its fields can include title, contents, URL, and so on. Different types of Field control which field you should index and which you should store with the index. The second and third lines add two fields to the document. Each field contains a field name and the content. This example adds two fields named "content" and "path", which store the content and the path of the text file, respectively. The last line adds the prepared documents to the index.

IndexWriter then does scoring (explained later) and stores in data structure like hashtable.

Scoring

The scoring in Apache Lucene is dependent on 3 factors-frequency, inverse document frequency, and field-length norm. Together, they are used to calculate the weight of a single term in a particular document at time of indexing.

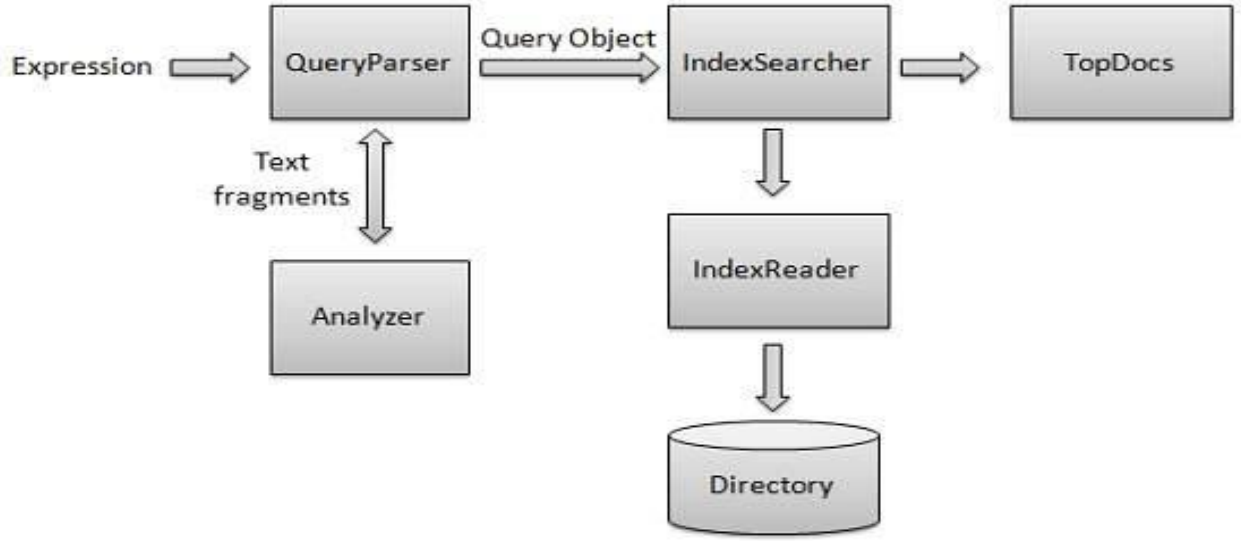
Frequency: How often does the term appear in this document? The more often, the *higher* the weight. A field containing five mentions of the same term is more likely to be relevant than a field containing just one mention.

Inverse Document Frequency: How often does the term appear in all documents in the collection? The more often, the *lower* the weight. Common terms like and or the contribute little to relevance, as they appear in most documents, while uncommon terms like elastic or hippopotamus help us zoom in on the most interesting documents.

Field-length Norm: How long is the field? The shorter the field, the *higher* the weight. If a term appears in a short field, such as a title field, it is more likely that the content of that field is *about* the term than if the same term appears in a much bigger body field.

Searching

Following is the flow of searches.



Searching Process

Classes Used:

1. QueryParser class parses the user entered input into lucene understandable format query. Initialize the QueryParser object created with a standard analyzer having version information and index name on which this query is to run.
2. IndexSearcher class acts as a core component which searcher indexes created during indexing process. Create a lucene directory which should point to location where indexes are to be stored.
3. IndexReader where we give Index directory as the input.
4. To start search, create a Query object by parsing search expression through QueryParser. Make search by calling IndexSearcher.search() method.
5. Top results are returned first.

3. Project Planning

Task Divisions:

Initial research on Integrating Crawler to project: Dhawal Rank
Developing createIndex() method: Dhawal Rank, Siddharth Khobare
Developing searchIndex() method: Shehraj Gupta, Mustafa Ali Misri
HTML to Text Parser: Shehraj Gupta, Mustafa Ali Misri
Reports: Dhawal Rank, Siddharth Khobare
Presentations: Dhawal Rank, Siddharth Khobare, Shehraj Gupta, Mustafa Ali Misri

4. Future Enhancements

We are trying to integrate the crawler in our project which will make our search engine dynamic searching the query online. The only drawback will be we cannot crawl the whole web as it has billions of pages. Also the index file will be too huge that we will require servers like Google. So for starters we will crawl only one domain. We also are in progress of developing a cool interface for the users using Java Server Pages.

5. Conclusion

In a nutshell, when Lucene indexes a document it breaks it down into a number of terms. It then stores the terms in an index file where each term is associated with the documents that contain it. You could think of it as a bit like a hashtable. When a query is issued it is processed through the same analyzer that was used to build the index and then used to look up the matching term(s) in the index. That provides a list of documents that match the query.

6. References

Apache Lucene Core, retrieved from https://lucene.apache.org/core/
Apache Lucene Scoring, retrieved from https://lucene.apache.org/core/3_5_0/scoring.html
Indexing Process Image, retrieved from http://www.tutorialspoint.com/lucene/images/indexing_process.jpg
Searching Process Image, retrieved from http://www.tutorialspoint.com/lucene/images/indexing_process.jpg