Department of Computer Science and Engineering

PROJECT REPORT

Text Search Engine
Name-Lovedeep Singh SID-16103104
Advisors- Dr. Rajesh Bhatia (Mentor) Prof. Mayank Gupta (Mentor) Prof. Anoop Dobhal Prof. Manish Kamboj

Declaration

I solemnly declare that the report TEXT SEARCH ENGINE is an outcome of our own work put throughout the duration of the fifth semester done under the able guidance of my advisors.

I further certify the following:

I. The work contained in this report is original and has been done by us under the general guidance of my mentors.

II. The work has not been submitted to any other Institution for any other degree/diploma/certificate in this university or any other University.

III. All the guidelines provided by the university in writing the report have been followed.

IV. Whenever I have used materials from other sources, I have given due credit to them by giving their details in the references.

Name: Lovedeep Singh
SID: 16103104

Advisors- Dr. Rajesh Bhatia (Mentor)
Prof. Mayank Gupta (Mentor)
Prof. Anoop Dobhal
Prof. Manish Kamboj

Abstract

Wikipedia contains useful information in various domains. It in itself is a good knowledge base to refer for a particular topic of interest. We go through Wikipedia for a number of articles whenever required. We simply search for the required information on Google and it gives us a number of results, including relevant results from Wikipedia. The idea is to create a Search Engine that specifically works on the Wikipedia Corpus.

Acknowledgement

I would like to express my great gratitude to my supervisor Prof. Mayank Gupta for his kind advice throughout the project. I also express my gratitude to Dr. Rajesh Bhatia, Prof. Anoop Dobhal and Prof. Manish Kamboj for guiding me in between whenever required.

Table of Contents

List Of Figures

List of Tables

CHAPTER 1
INTRODUCTION AND BACKGROUND

Introduction

Search engines are programs that search documents for specified keywords and returns a list of the documents where the keywords were found. A *search engine* is really a general class of programs, however, the term is often used to specifically describe systems like Google, Bing and Yahoo! Search that enable users to search for documents on the World Wide Web.

Web Search Engines:

Typically, Web search engines work by sending out a *spider* to fetch as many documents as possible. Another program, called an *indexer,* then reads these documents and creates an index based on the words contained in each document. Each search engine uses a proprietary algorithm to create its indices such that, ideally, only meaningful results are returned for each *query*.

As many website owners rely on search engines to send traffic to their website, and entire industry has grown around the idea of optimizing Web content to improve your placement in search engine results.

Common Search Engines:

In addition to Web search engines other common types of search engines include the following:

Local (or offline) Search Engine:

Designed to be used for offline PC, CDROM or LAN searching usage.

Metasearch Engine:

A search engine that queries other search engines and then combines the results that are received from all.

Blog Search Engine:

A search engine for the blogosphere. Blog search engines only index and provide search results from blogs (Web logs).

Background

Types Of Search:

Search can be categorized in two ways:

1. Exact values
2. Full text

Exact values:

Exact value Foo is not same as foo. The exact value 2014 is not same as 2014-12-03. Exact values are easy to query. The *decision is binary*, a value either matches the query or it doesn't.

Ex: SELECT * FROM student WHERE name = "Ravi".

Full text:

Querying full text data is much more subtle. We are not asking "Does this document match the query?" but "How well does this document match the query?" means *how relevant is this document to the query*.

Full-text search engines evolved much later than traditional database engines, as corporations and governments found themselves with more and more unstructured textual data in electronic format. These new text documents didn't fit well into the old table-style databases, so the need for unstructured full-text searching was apparent.

Since it was developed later, search engine technology borrowed heavily from the database world, and many search engines still employ some type of traditional table structures in their underlying architecture. Some text retrieval companies were even staffed with employees who came from traditional database company backgrounds. Many of the key RDBMS paradigms have also migrated into search engine technology, though often renamed or recast.

Overview of Full-Text Architecture

Figure 1 shows a generalized, high-level architecture of most of the commercial quality full-text engines currently available.

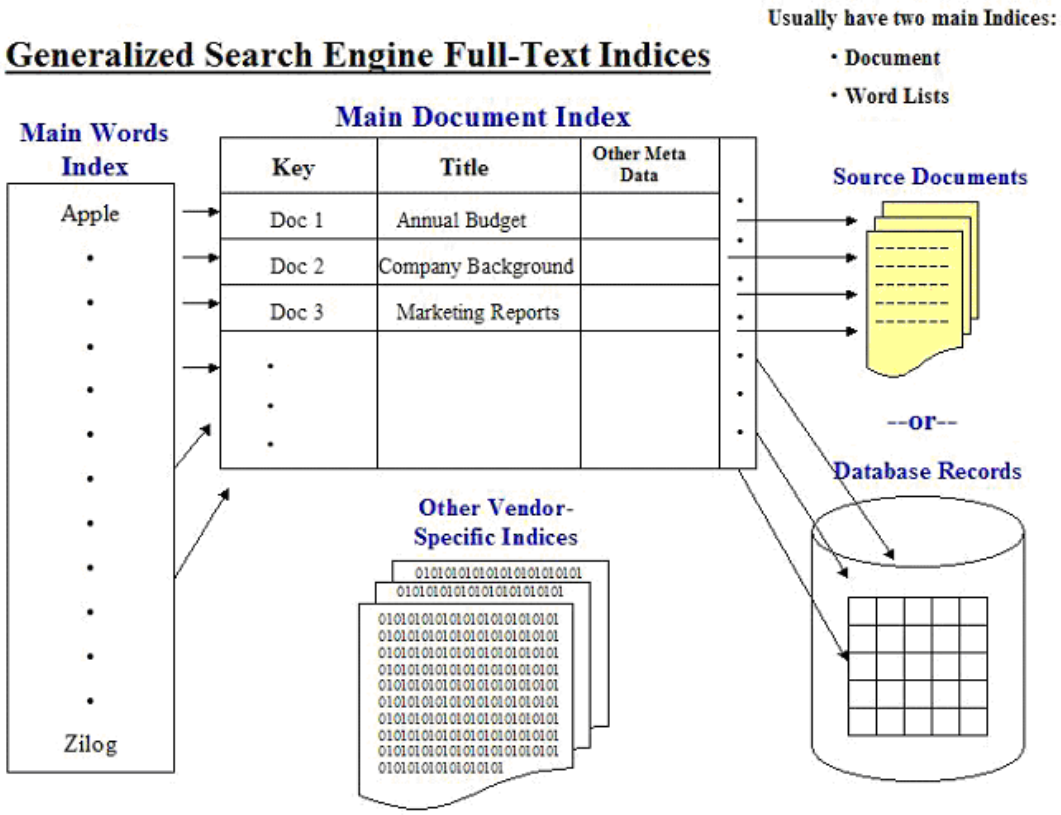

Figure 1
Sources:

Last Updated Jan 2009

By: Miles Kehoe, New Idea Engineering, Inc., Volume 2 - Issue 1 - June 2004

http://www.ideaeng.com/database-full-text-search-0201

The indexing process begins when an application inserts data into a row in the main document index. In the simplest case, this index contains one row per document, and at a minimum contains the name of the external file or document stored in the key field. Additional field values – for example, document title – can be inserted at the same time. If the source of the data resides in a relational database, the primary key in the relational table or view goes in the main document index key field.

Once the data is inserted, the indexing engine opens the external document and creates an ordered word list to load into the main word index. The engine repeats this process for every record in the document index.

While searching is generally available during the indexing process, only completed records are searchable; and for performance, most engines batch together a number of records to index more efficiently.

When a query arrives, either programmatically or as a result of a user request, the full-text engine accesses the sorted and optimized word index to identify which documents contain the requested term(s). The engine creates a list of documents that qualify, typically provided as a list of pointers into the main document index.

This permits the engine to access and display a result list made up of any fields stored in the main index, calculate a relevance weight, and display a list of results.

The description of the internal architecture will be extended throughout the report.

Advantages Over Traditional Techniques

This section summarizes some of the advantages of search engines over traditional database engines. These advantages are typical of high-end search engine products.

- Optimized to Handle Textual Data

Full-text engines are specifically optimized to handle textual data. In particular, proper customer names, city, street names and other geographic markers are all textual in nature. This type of textual data is often subject to multiple valid and invalid spellings. Further, proper names are often represented in many un-normalized forms.

- Granular Index Structure

Index structure of full text engines is more granular, allowing for rapid indexed access to specific words and phrases.

- More Flexible Query Operators

Search engines offer many more query operators, such as language stemming, thesaurus, fast wildcard matches, statistically based similarity and word densities, proximity, and other "fuzzy" matching techniques.

- Hybrid Search Support

Search engines allow for hybrid searches that can search both traditional fielded data and textual data in the same query.

- Relevancy Weighting

Search engines provide relevancy weights to likely matches, allowing for much finer tuning of search results.

Technical Similarities

While relational database systems and full-text search engines are optimized to process fundamentally different types of data, there are a number of similarities between the two.

- Query Processing

For both relational and full-text engines, a query is processed and passed to the engine to retrieve data. While most relational systems use largely standard syntax, full-text systems generally use proprietary syntax. One notable exception is Fulcrum, which bases its query syntax on SQL.

- Data Loading

As with relational systems, data must be loaded into full-text systems. And like relational systems, how the data is organized and loaded can impact later success of the application. And, as with relational systems, data is loaded into full-text engines both in bulk periodically or on a record-by-record real-time basis.

- Data Indexing

Once data is loaded into relational systems, it should be indexed for optimum performance. The same is true for full-text systems, although indexing is required for any retrieval, not just to optimize performance.

Search engines create much more extensive indices than a traditional relational systems. A full-text engine typically creates an index of every word in every document, and some full-text engines index character pairs and triplets as well as full words.

Technical Differences

While there are similarities between full-text and relational technologies, there are a number of differences as well because of the fundamental differences between the types of data being indexed and the flexibility of the retrieval options. While the differences can present some challenges, they also present the opportunity to take advantage of the key features of full-text search to provide an innovative solution to the problem at hand.

- Differences in Technical Vocabulary

As mentioned, full-text and relational systems use many similar terms. For example, both relational and full-text systems use the term "index" in almost the same way, but there are some differences. These vocabulary differences are of special note for people with a relational background who start working with full-text engines.

As suggested earlier, a user or program generates a query that is passed to a relational or full-text engine, which processes the query and returns a result set. A relational system returns rows of fielded data. A full-text system returns meta-data representing documents.

A database or table in the relational world is similar, but not identical to, a collection in the full-text world. In some cases, a collection is more analogous to a relational view. What a relational system refers to as a result-set record is often referred to as a document by full-text engines.

Other terms are specific to full-text engines and vendors, and have no meaning in the relational environment.

In the same way, some concepts from the relational world are conspicuously absent from full-text environment. As we will discuss, full-text engines do not directly use joins and outer joins. Also, since full-text engines tend to serve as a read-only medium, there is much less emphasis on transactions and related mechanisms. People familiar with relational systems may be initially puzzled by their absence, given their importance in the database world.

- Data Structure

Relational engines typically store structured data such that associated data is stored in the same record or row, with the components organized into identified fields or

columns of information. Formatted documents and images are maintained in special long fields such as binary long objects, although the scope of commands that can be used on these long fields is limited.

Full-text engines are optimized for processing formatted and unformatted documents, but they must also be able to process limited structured data such as document titles, authors, and descriptions. The command sets of most full-text engines are optimized for flexible retrieval of documents, with a less feature-rich set of commands to process and retrieve the structures data.

- Query Syntax

There is much wider variety of query language syntax in full-text engines. Many engines do support the classic "AND", "OR" and "NOT" operators in some form; some engines even allow for complex nested queries using parenthesis or other nesting syntax. A modern pseudo-standard is the "Internet syntax", which allows for "+" and "-" to be used as a shorthand for Boolean operators. Many search engines actually support multiple syntaxes that can often be configured when a search is performed.

- Additional Full-Text Operators

Advanced search engines typically offer a wider variety of query operators than traditional relational engines. Examples include thesaurus, language stemming, "typo" and proximity qualifiers.

While some of these operators could be simulated with a relational database, others require special indexing of a scope and granularity well beyond that which is available in relational systems.

- Weighting

Relational queries typically return rows that match the specified query in arbitrary order, or sorted by a field specified in the query. Records that are returned match the query; and records that do not match the query are simply not returned.

Most advanced full-text engines will also retrieve only those documents that match the query, but additionally "weight" each returned document with a relevancy score, so that the results set is not just an unordered list of matching records. Typically, the weight is calculated using proprietary algorithms based on the vendor, although most

engines provide syntax for affecting the final weight. Full-text queries typically return a much larger percentage "records" than a traditional query, but frequently only the higher ranking matches are of interest.

Most full-text result lists are intended for presentation to a human operator, who can evaluate attributes of the returned document – title, author, summary, or other attributes – to identify the most appropriate document.

- Different Usage Patterns

Search engine users typically issue shorter initial queries than a skilled SQL user, but will often issue subsequent refined queries to drill-down to find the most relevant documents.

- Joins

Search engines typically do not perform database joins; instead they often have a much simpler arrangement of data, perhaps somewhat analogous to relational database views.

- Outer Joins

There is no direct analog to an "outer join" in full-text engines. Any required complex data gathering tasks would usually be performed prior to indexing the full-text data

- Simpler Table Structure in Full-Text Systems

Unlike traditional databases, a search engine administrator will typically deal with only a few document tables, referred to by most vendors as "collections", "catalogs" or "document indices". Nonetheless, full-text engines may internally be using multiple table structures to store their indices.

A hypothetical Human Resources illustration may help clarify the difference between relational databases and full-text indices.

With a relational database, HR data would be broken up into many tables. Even for a simple object model such as "employees", there may a dozen tables, relating addresses, referencing department records, tax and payroll records, and 401K records.

For a full-text engine, employees would probably be indexed in a single collection, with each "document" representing an individual employee. This document can

represent a single, physical document, or perhaps more likely in this HR example, a virtual document.

- Virtual Documents

At the heart of full-text systems is the concept of a “unit of retrieval”. This is often a file or document, but when the source data is relational in nature, the unit of retrieval is typically a “virtual document” composed of data from a number of columns and tables.

In the earlier example of an HR database, a virtual document made up of a number of columns is created for each employee. This could be an actual relational view, or it could be a ‘virtual view’ created only during the full-text indexing phase. This primarily textual document might have the employee name, job skills, manager's name, and perhaps a copy of the employee’s resume. A full-text search for the employee name, the manager name, or for the skills listed in the employee’s resume would be returned. Note that a full-text collection would probably not be used for reporting purposes, or to find tax or 401K-contribution information.

Virtual documents often contain predefined regions known as “zones”, which are similar to fields. With a zone, created at index time, searches can be narrowed down more precisely. Using the earlier HR example, if the manager's name is identified as a zone at index time, a query could request only those documents where the manager’s name appears in the manager zone. Thus, a document with the manager’s name in the employee resume would not be returned in this case.

- Document Keys

As in relational systems, full-text documents are identified using a key field. In full-text systems, the document key may be a file or document name, or a URL. If the full-text system is used to index and search relational data, the document key is typically the primary key in the relational table or view.

- Data Types and Document Formats

Most full-text engines support character, numeric and data/time formats in addition to text. These fields can be used in searching, sorting and displaying results. Their use in searching is discussed below.

There is a much wider variety of native document format support with full-text engines. These documents are not just stored as "blobs"; they are actually opened and their contents indexed. Typically supported formats include text, HTML, and various 'binary' document formats such as Microsoft Word, Excel, and PowerPoint; some engines handle PDF and XML as well. Many search engines leave the original documents intact, in their original location, and store binary indices pointing to those documents.

Vocabulary Comparison Summary

Table 1 summarizes vocabulary between relational and full-text databases. Due to the large number of vendors and broad use of terms, this table can only serve as an approximation.

Table 1

RDBMS Term(s)	Full-Text Term(s)	Notes
database	collection, document index or catalog	Varies widely
table	segment or partition	This is typically transparent to the casual Search Engine administrator; you typically do not address individual partitions or segments.
record	document, record, page, web page or result	Traditional search engines deal in terms of "documents"; more modern Internet engines talk of "web pages"
field	field, document field, meta field, zone	Search engines often have two different ways of storing data. When it is stored in the document index, it is usually called a "field". When it is stored in the word index, it is usually called a "zone". Each type of storage has its own benefits
blob	zone	Larger segments of text are typically stored as zones
index (noun)	collection, document index or word index	In both worlds it typically refers to a large binary data-store residing on a disk
index (verb)	index or spider	The tabulating and storing of data into the binary indices
query	query or search	Same terminology in both
join	n/a	Full-text engines do not usually do "joins" at search time
import/export	n/a	Most full-text engines do not offer robust important and export capabilities. Some vendors do offer import tools. Though indexed, documents are typically not imported directly into a full-text database. The process of "indexing" or "spidering" can be thought of as a type of import, although the original source documents are left where they were.
SQL	n/a	Though there are some full-text query language standards, they are not widely supported or implemented. The closest semi-standard is the "Internet syntax" of some vendors, where + and - service as AND and NOT, quotation marks demark exact phrases, and ()'s are often recognized to convey precedence.
ODBC	n/a	There is no widely used standard. Most modern full-text engines do offer access via the HTTP protocol's CGI mechanism, though the specific field names to use vary widely from vendor to vendor.

Source:
Last Updated Jan 2009

By: Miles Kehoe, New Idea Engineering, Inc., Volume 2 - Issue 1 - June 2004

http://www.ideaeng.com/database-full-text-search-0201

CHAPTER 2
MOTIVATION AND PROBLEM FORMULATION

Motivation

Ground motivation:

Wikipedia contains useful information in various domains. It in itself is a good knowledge base to refer for a particular topic of interest. We go through Wikipedia for a number of articles whenever required. We simply search for the required information on Google and it gives us a number of results, including relevant results from Wikipedia. The idea is to create a Search Engine that specifically works on the Wikipedia Corpus.

Additional Motivating factors:

- Opportunity to peek in immense world of search engines
- Moreover search has been one of the fundamental problems in computer science domain
- To work in JAVA

Problem Formulation:

Problem Definition

The aim of this project is to build a prototype of a search engine, which works on millions of Wikipedia pages (which are in XML format) and retrieves the top 10 relevant Wikipedia documents that match the input query. This search engine takes Wikipedia corpus in XML format, which is available at wkipedia.org as input. Then it indices millions of Wikipedia pages involving a comparable number of distinct terms. Then given a query, it retrieves relevant ranked documents and their titles using index.

Objectives

The objective of this project is to develop a search engine for Wikipedia. This search engine is named Text Search Engine. It has following responsibilities:

- Maintains an inverted index.
- Provides keyword search.
- Ranks Wikipedia pages as per relevancy.

Scope of the project

The following things will be provided:

- Technical Review: Review areas will be concepts in the search engine world, components of the search engine, and methods of search engine.
- System Development: This encompasses the important phases of the project. Context level design, System level design, java program of search engine.
- Documentation: This is the final project report. It contains technical documents such as DFDs, Use case diagrams, activity diagrams etc.

The following things will not be provided:

- User interface design: User interface is not the focus of this project.
- Security module: Security module will not be included in the development of the system. The security inbuilt in java is enough when apply to this system. Since there are no transactions involved in this system, very secure system is not required.

Development Plan and Schedule

Major Technical Activities:

The development will follow a typical development life cycle format.

- System Analysis: An analysis of search engine methodologies present and selection of suitable method.
- System Design: Starting from high level moving to lower level details, using DFDs and UML development techniques.
- Systems Development: Coding in java using IntellijIdea as IDE
- Systems Testing: Final systems testing and acceptance testing will be performed.

CHAPTER 3
REQUIREMENTS AND DESIGN

Requirements Definition

In this section, requirements definition is given.

Functional Requirements

Function requirements mean the physical modules, which are going to be produced in the proposed systems. The functional requirements of the proposed system are described below:

Words extraction:

The system will provide string tokenizer. It will take as input wikipage in XML format and parse it and extract words out of the wikipage. SAX parser is used for parsing the wikipage.

Building an Inverted Index:

After extraction, it will build an inverted index out of it. It will build 26 index files corresponding to each alphabet. One index file of all alphabets will also be created. This will be later used for searching the query input by the user.

Ranking:

On searching a particular word, the system will list the matching documents in order of rank as per the relevancy.

Searching:

Given a query, system will retrieve relevant ranked documents and their titles using inverted index.

Non-Function Requirements

Non-Function requirements mean the characteristics that are not related to system's physical functions but are the characteristics that favorite to the physical functions.

User Friendly:

The system should be easy to use, i.e. user friendly, for both administrator and external users. This can be done for providing function descriptions.

DESIGN:
CONTEXT LEVEL DFD:

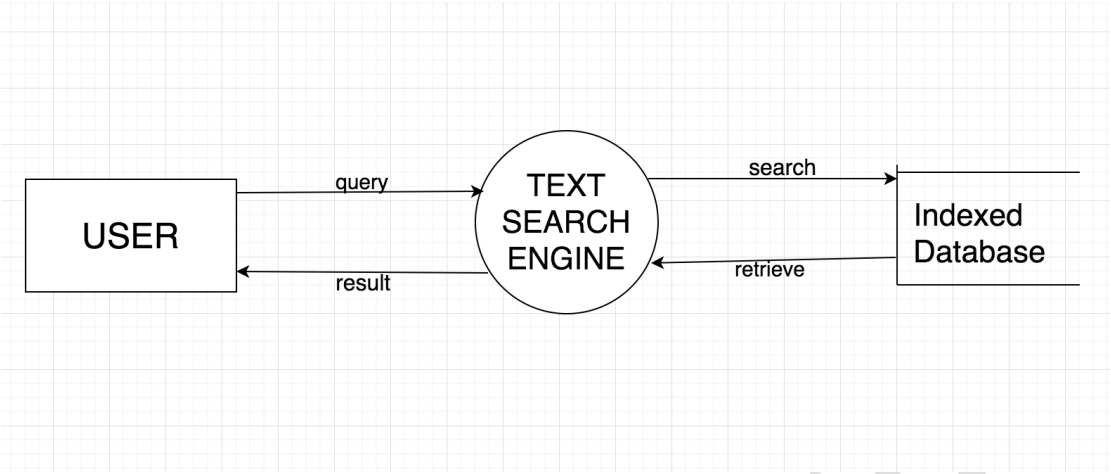

Figure 2

CREATION OF INDEXED DATABASE:
METHOD DIAGRAM:

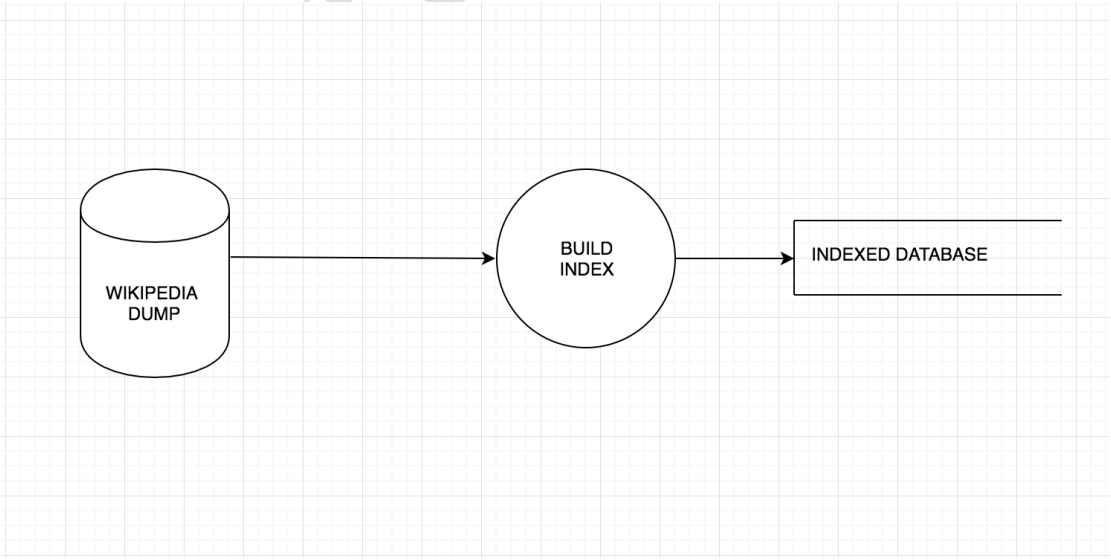

Figure 3

METHOD DIAGRAM FOR BUILD INDEX

Module 1:

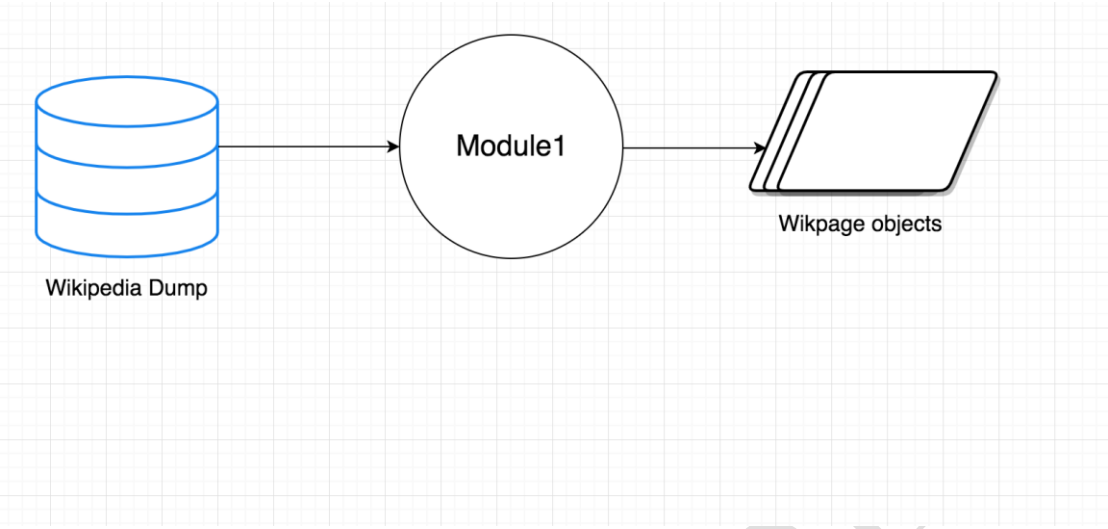

Figure 4

Module 2:

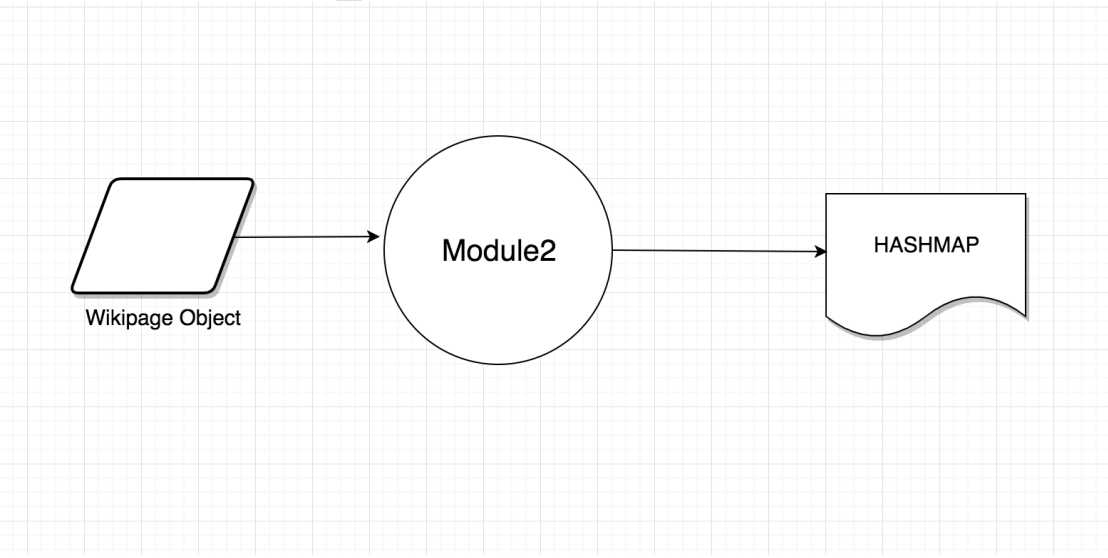

Figure 5

Module 3:

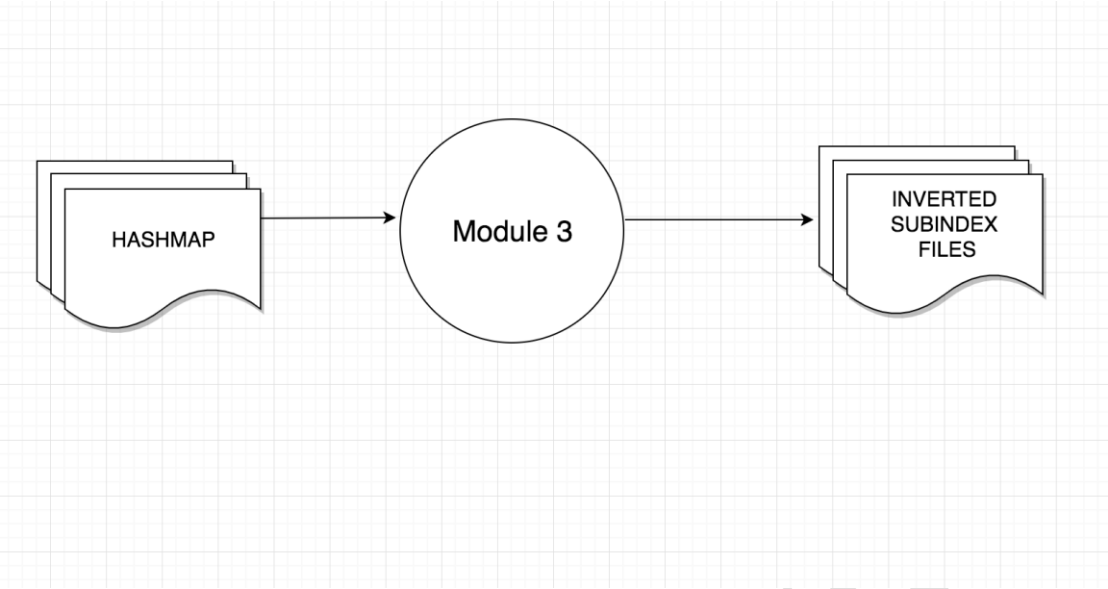

Figure 6

Module 4:

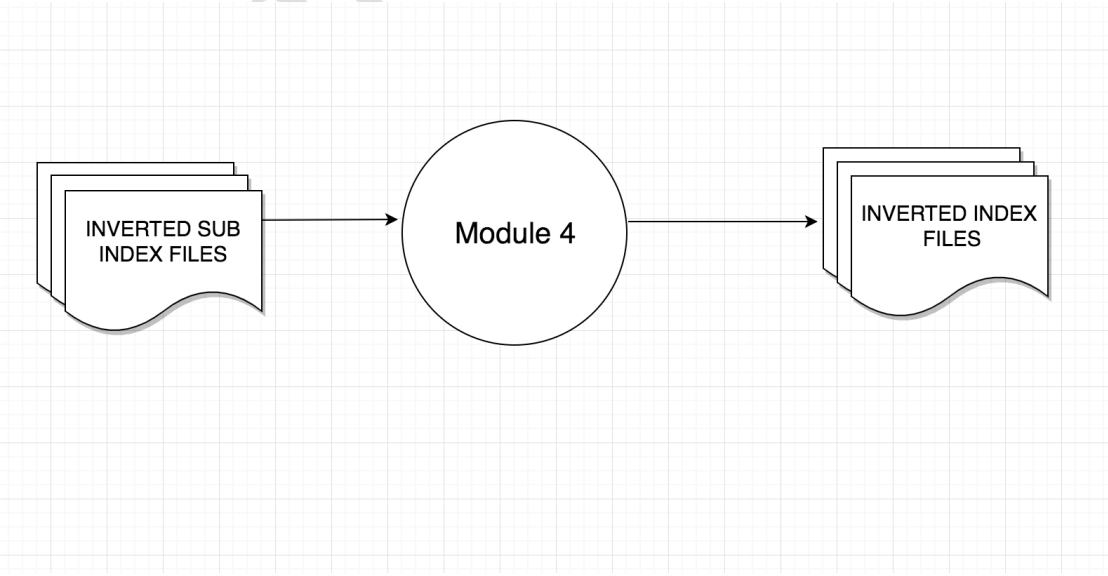

Figure 7

Module 5:

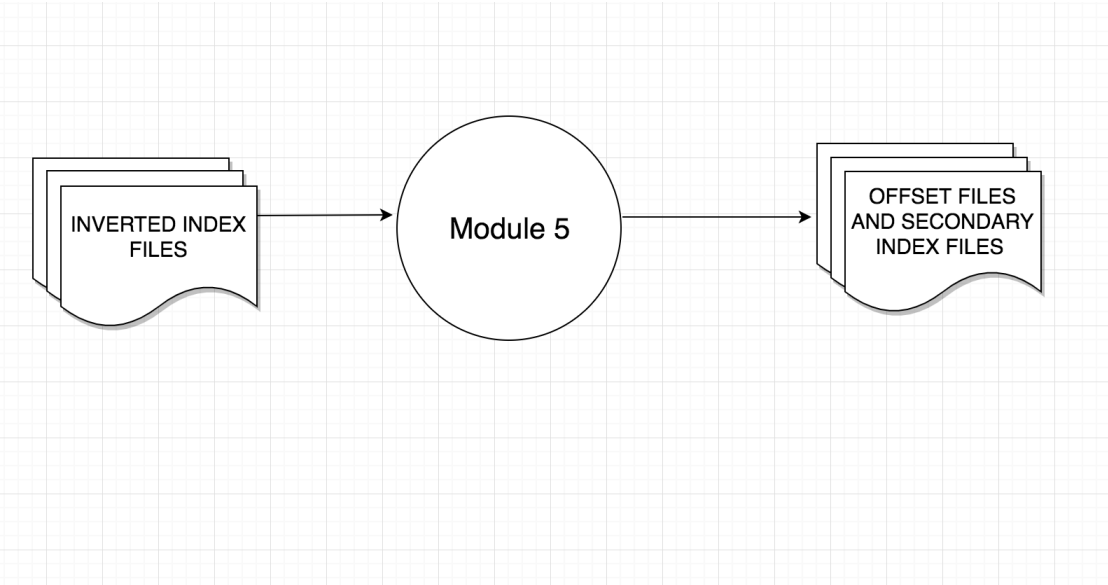

Figure 8

USER USE CASE VIEW

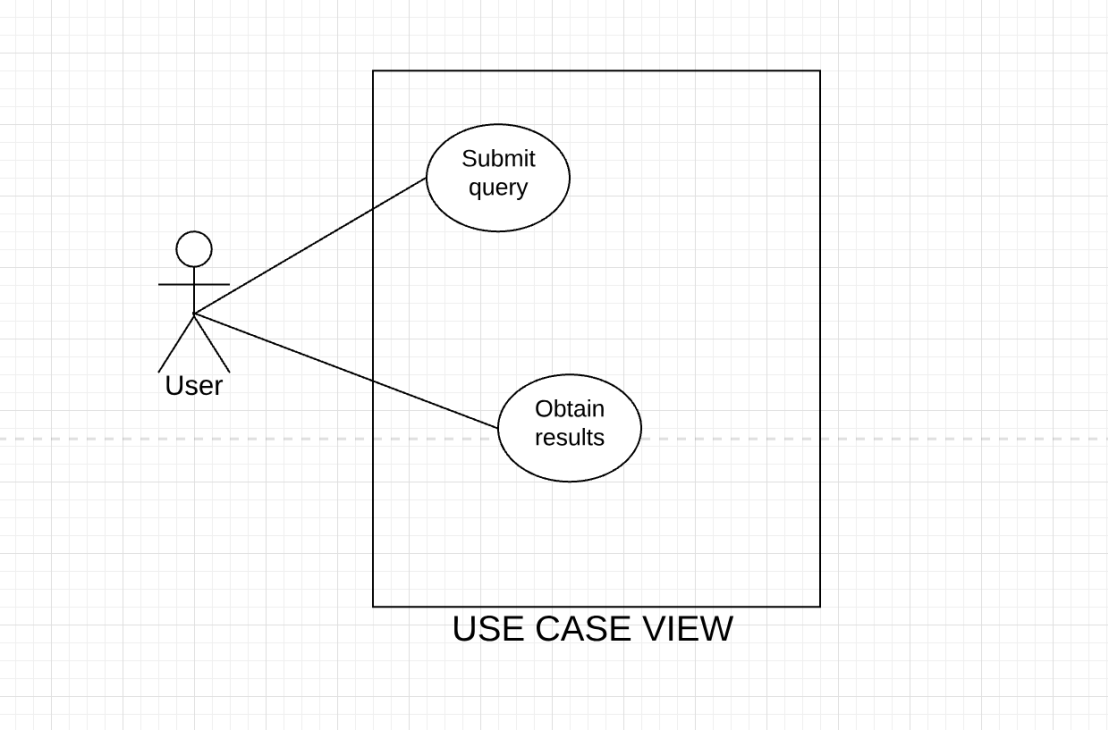

Figure 9

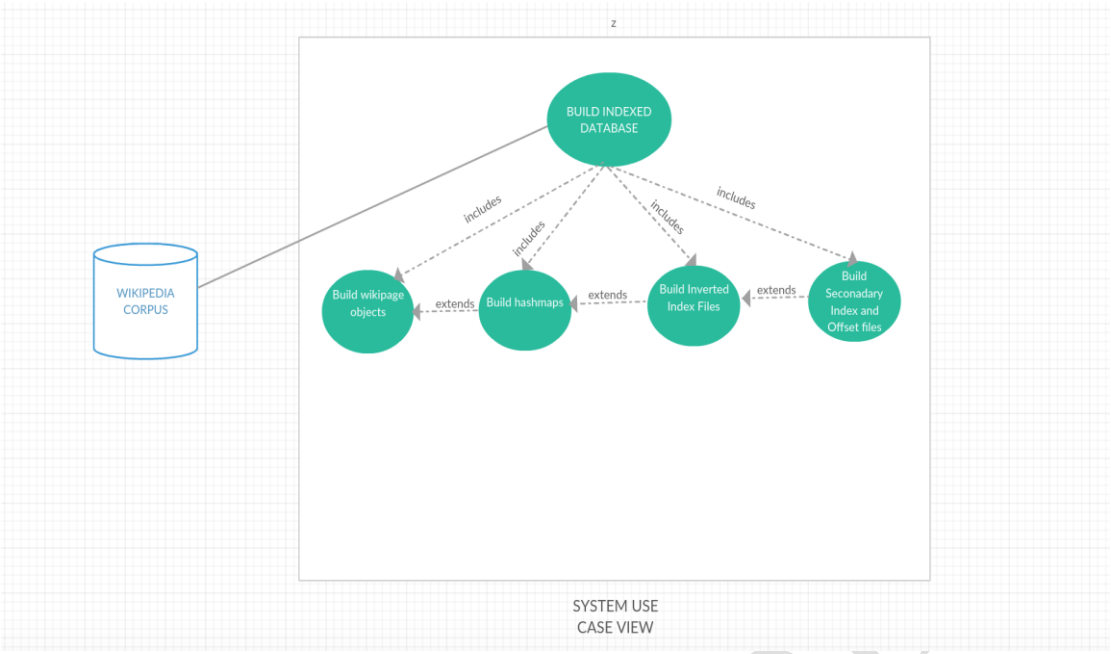

Figure 10

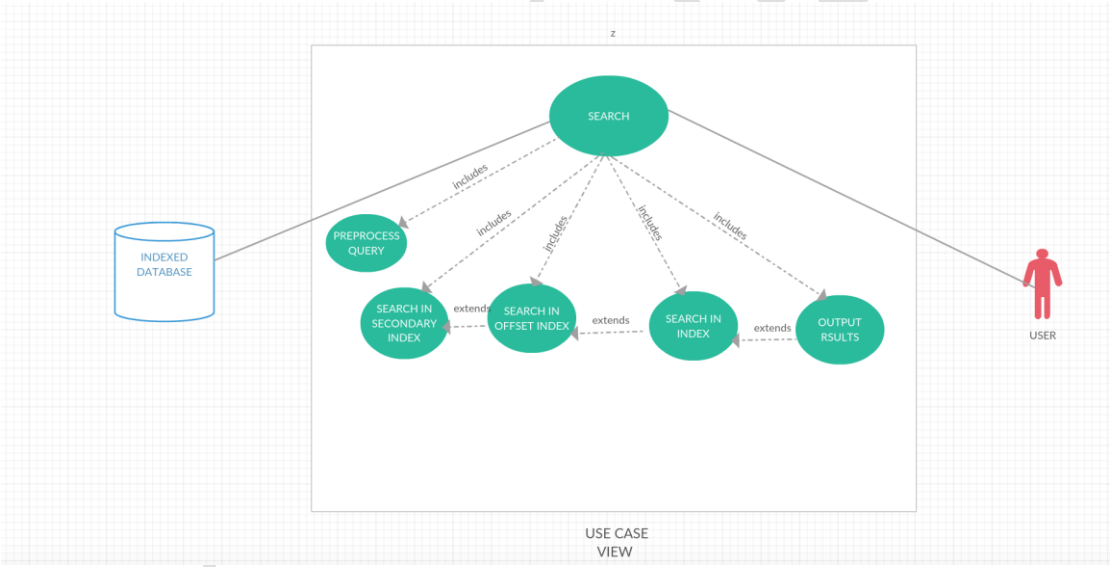

Figure 11

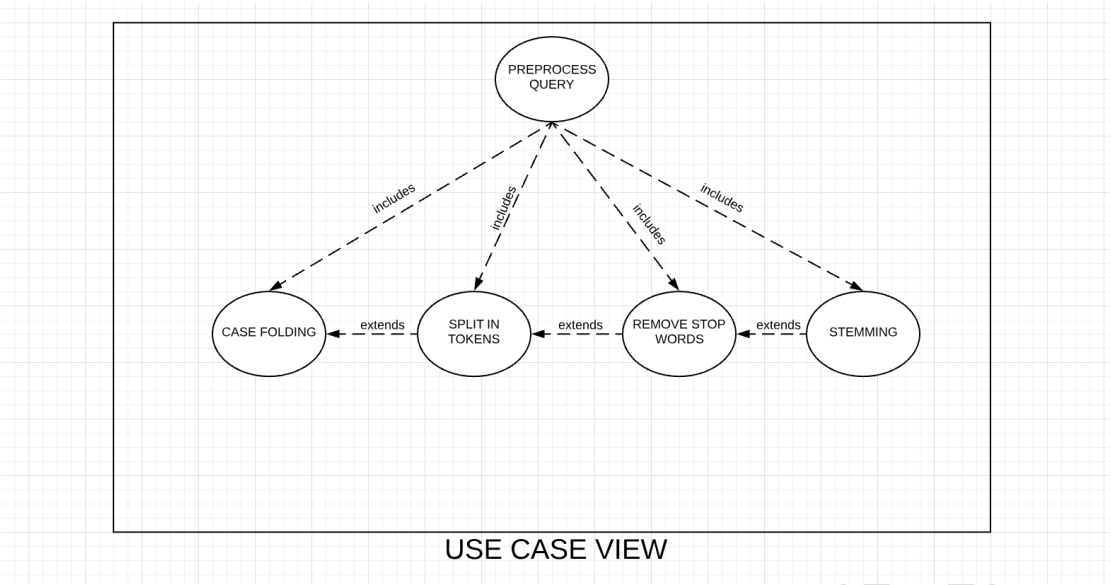

Figure 12

ACTIVITY DIAGRAM:

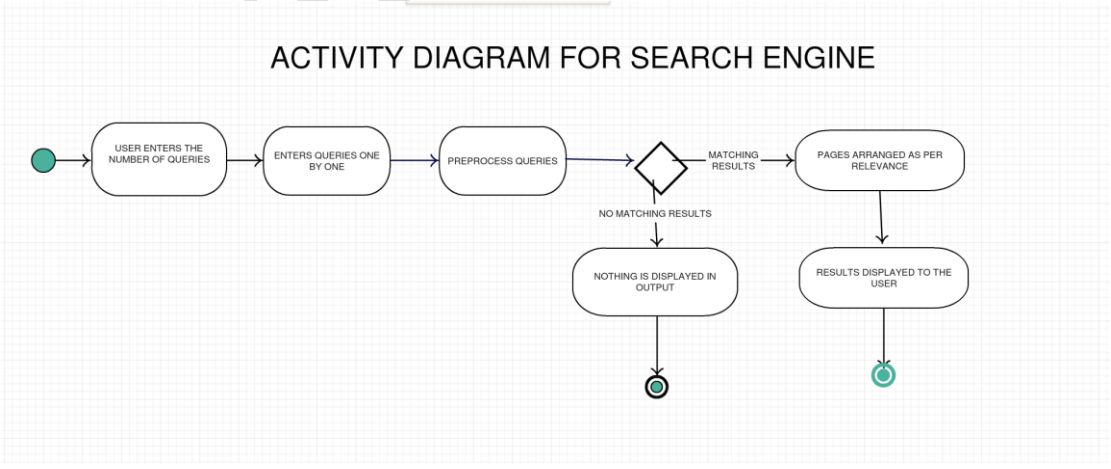

Figure 13

SYSTEM LEVEL DFD

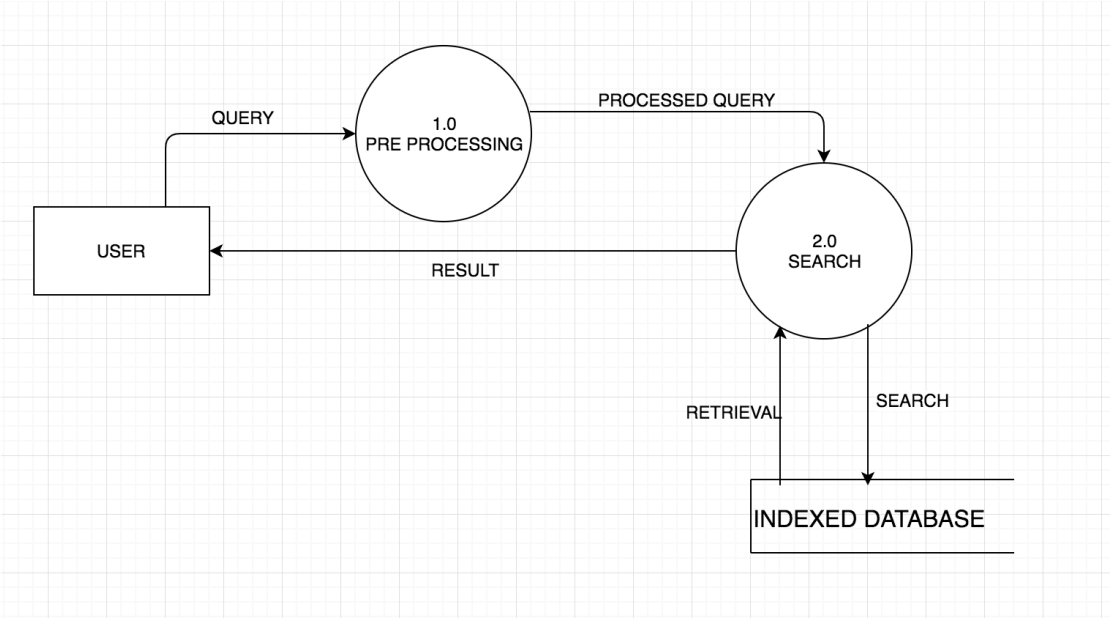

Figure 14

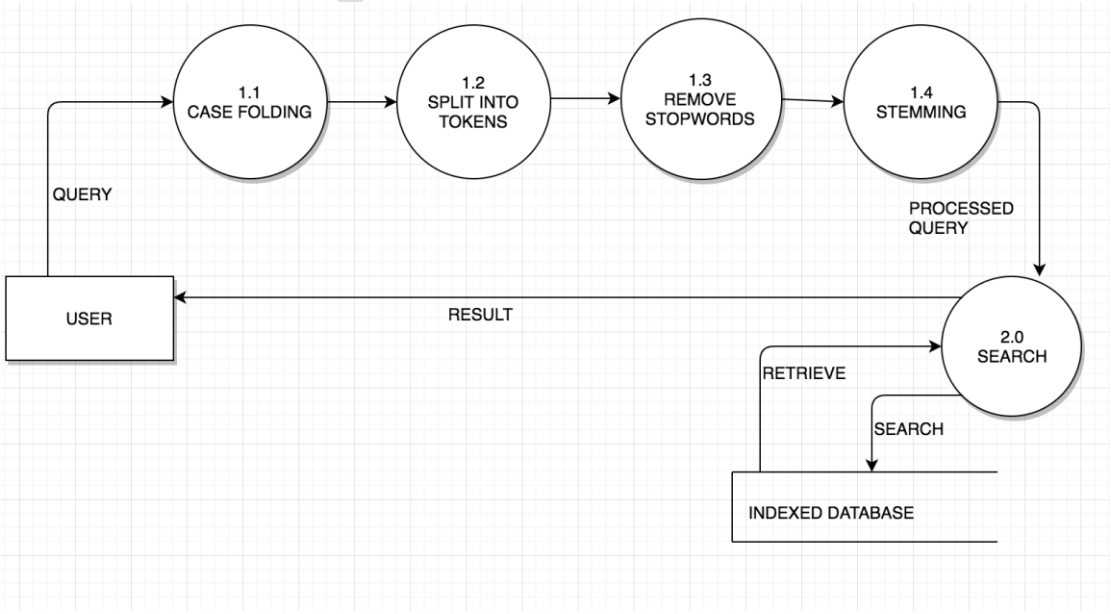

Figure 15

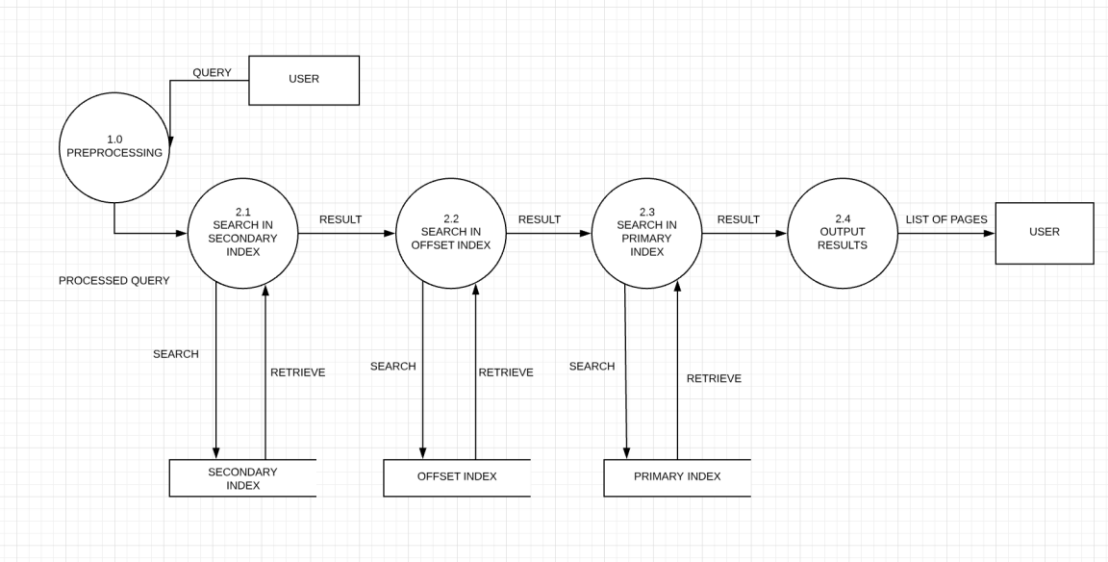

Figure 16

CHAPTER 4
IMPLEMENTATION DETAILS

First, a brief understanding on the concepts applied in the development of the project is given. Then the actual implementation is outlined.

Inverted Index:

To facilitate full text search, in our project we first analyze the text and then use the result to build inverted index.

An inverted index contains all the unique words that appear in any document, and for each word, a list of document in which it appears.

To create inverted index, we first split the content field of each document in separate words (which we call tokens), create a sorted list of all the unique terms, and then for each term, we maintain a list of documents in which it is present.

Text Preprocessing:

1. Tokenization
2. Sentence Segmentation
3. Case Folding
4. Removing Stop words
5. Stemming

Before making any inverted index, the Wikipedia page is preprocessed.

Tokenization:

Splitting text into individual words

Sentence segmentation:

Identifying end of a particular sentence

Case folding:

Ex: GnU = gnu

Everything is mapped to lower case

Removing Stop words:

Removing words like is, an, a, the, etc. These are not important coming to search.

Stemming:

Obtaining root word from a given word is stemming.

Ex: Jumping → jump

We've used Porter Stemming Algorithm in this project.

XML (Extensible Markup Language)

Defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality, and usability across the Internet. It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures such as those used in web services.

KEY TERMINOLOGY

This is not an exhaustive list of all the constructs that appear in XML; it provides an introduction to the key constructs most often encountered in day-to-day use.

Character

An XML document is a string of *characters*. Almost every legal Unicode character may appear in an XML document.

Processor and application

The *processor* analyzes the markup and passes structured information to an *application*. The specification places requirements on what an XML processor must do and not do, but the application is outside its scope. The processor (as the specification calls it) is often referred to colloquially as an *XML parser*.

Markup and content

The characters making up an XML document are divided into *markup* and *content*, which may be distinguished by the application of simple syntactic rules. Generally, strings that constitute markup either begin with the character < and end with a >, or they begin with the character & and end with a ;. Strings of characters that are not markup are content. However, in a CDATA section, the delimiters <![CDATA[and]]> are classified as markup, while the text between them is classified as content. In addition, whitespace before and after the outermost element is classified as markup.

Tag

A *tag* is a markup construct that begins with < and ends with >. Tags come in three flavors:

- *start-tag*, such as <section>;
- *end-tag*, such as </section>;
- *empty-element tag*, such as <line-break />.

Element

An *element* is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start-tag and end-tag, if any, are the element's *content*, and may contain markup, including other elements, which are called *child elements*. An example is <greeting>Hello, world!</greeting>. Another is <line-break />.

Attribute

An *attribute* is a markup construct consisting of a name–value pair that exists within a start-tag or empty-element tag. An example is , where the names of the attributes are "src" and "alt", and their values are "madonna.jpg" and "Madonna" respectively. Another example is <step number="3">Connect A to B.</step>, where the name of the attribute is "number" and its value is "3". An XML attribute can only have a single value and each attribute can appear at most once on each element. In the common situation where a list of multiple values is desired, this must be done by encoding the list into a well-formed XML attribute[i] with some format beyond what XML defines itself. Usually this is either a comma or semi-colon delimited list or, if the individual values are known not to contain spaces,[ii] a space-delimited list can be used. Welcome! , where the attribute "class" has both the value "inner greeting-box" and also indicates the two CSS class names "inner" and "greeting-box".

XML declaration

XML documents may begin with an *XML declaration* that describes some information about themselves. An example is <?xml version="1.0" encoding="UTF-8"?>.

Ex:

```
<play>
<author> Shakespeare </author>
<title> Macbeth </title>
<chapter number = “vii” >
    <title> Macbeth’s castle </title>
</chapter>
</play>
```

Parsing:

Parsing is the problem of transforming a linear sequence of characters into a syntax tree. Nowadays we are very good at parsing. In other words, we have many tools, such as lex and yacc, for instance, that helps us in this task. However, in the early days of computer science parsing was a very difficult problem. This was one of the first, and most fundamental challenges that the first compiler writers had to face.

Parser:

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

We’ve used SAX (Simple API for XML) parser available in java library in this project. SAX is an event-based parser for XML documents. SAX is a streaming interface for XML, means the application-using SAX receives event notifications about the XML document being processed, an element and attribute at a time in sequential order starting at the top of the document. It reads and XML document from top to bottom, recognizing the tokens that make up a well formed XML document. Tokens are processed in same order that they appear in the document. Application program provides an event handler that must be registered with the parser.

Term Frequency:

It is a numerical statistic that is intended to reflect how important a word is to the document in a collection (or) corpus.

Ex: consider a query "borrow" and we wish to determine which document is relevant to the query.

1st approach: Eliminate all documents that do not contain "borrow". This still leaves many documents.

2nd approach: Count the number of times each term occurs in a document.

Weighted concept:

A word appearing once in title is not same as appearing once in body. The document in which it appeared in Title must have been more relevant. To incorporate this factor, weights are given to various domains of the Wikipedia page.

Title-1000

Infobox-20

Links-10

Category-50

Body-1

Term frequency = 1 + log (weighted sum).

Setbit :

Title	Infobox	Links	Catetory	Body
0 or 1	0 or 1	0 or 1	0 or 1	0 or 1

Varies from 0 to 31.

Disadvantages with term frequency:

Consider a query "the brown". Since the term "the" is so common term, frequency tends to incorrectly emphasize the document, which happen to use "the" more frequently. Solution to this problem is Inverted Document Frequency.

Inverted Document Frequency:

This diminishes the weight of terms that occur very frequently in the documents and increases the weight of terms that occur rarely.

idf = log ((total number of docs)/(Number of documents the word occurred))

Final Index contains:
word idf # docid1 – setbit : tf ; docid2 – setbit : tf ; docid3 – setbit : tf ;

Structure of Indexing:
26 index files corresponding to each alphabet is created. One allwords.txt index file is created which simply includes word and its total count in all documents. It is used to calculate idf.

Offset and Secondary Index:
To make search faster, for each alphabet's index file, two additional files offset.txt and secondary.txt are created. Secondary file contains some words in gaps and their byte number in offset file. Offset file contains each word and its byte number in index file. Index file contains the real index structure content.

IMPLEMENTATION

IntelliJIdea

The entire project is implemented using IntelliJIdea. IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software. It is developed by JetBrains (formerly known as IntelliJ), and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition. Both can be used for commercial development.

WIKIPEDIA CORPUS:

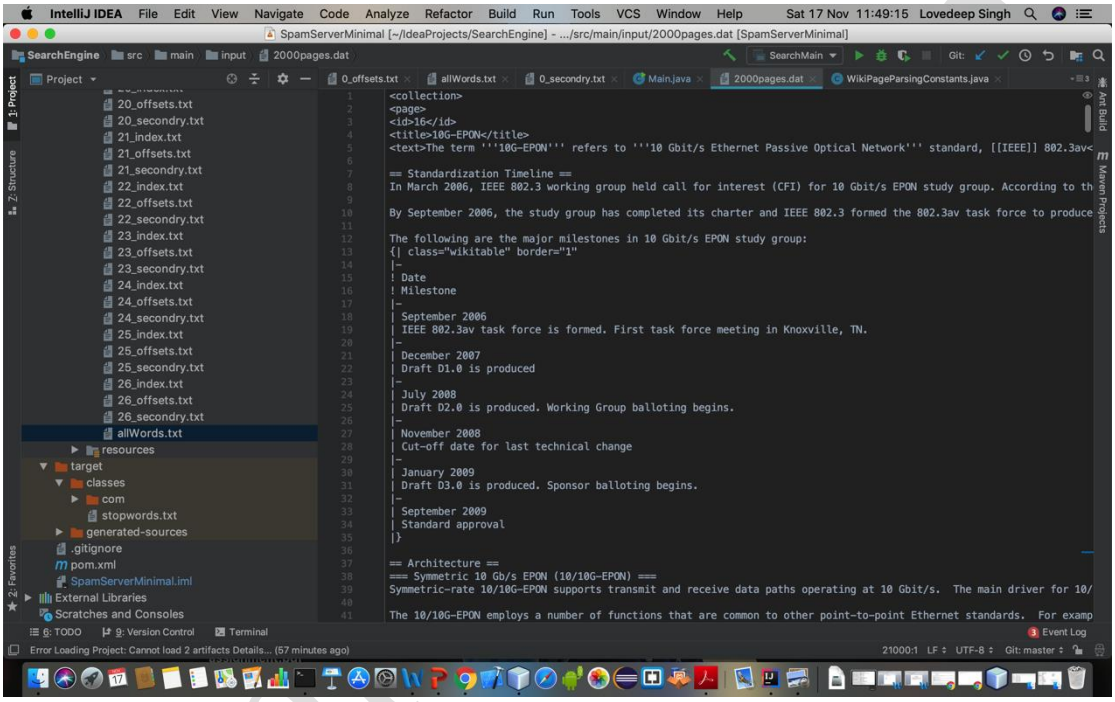

Figure 11

INDEX FILES:
allwords.txt

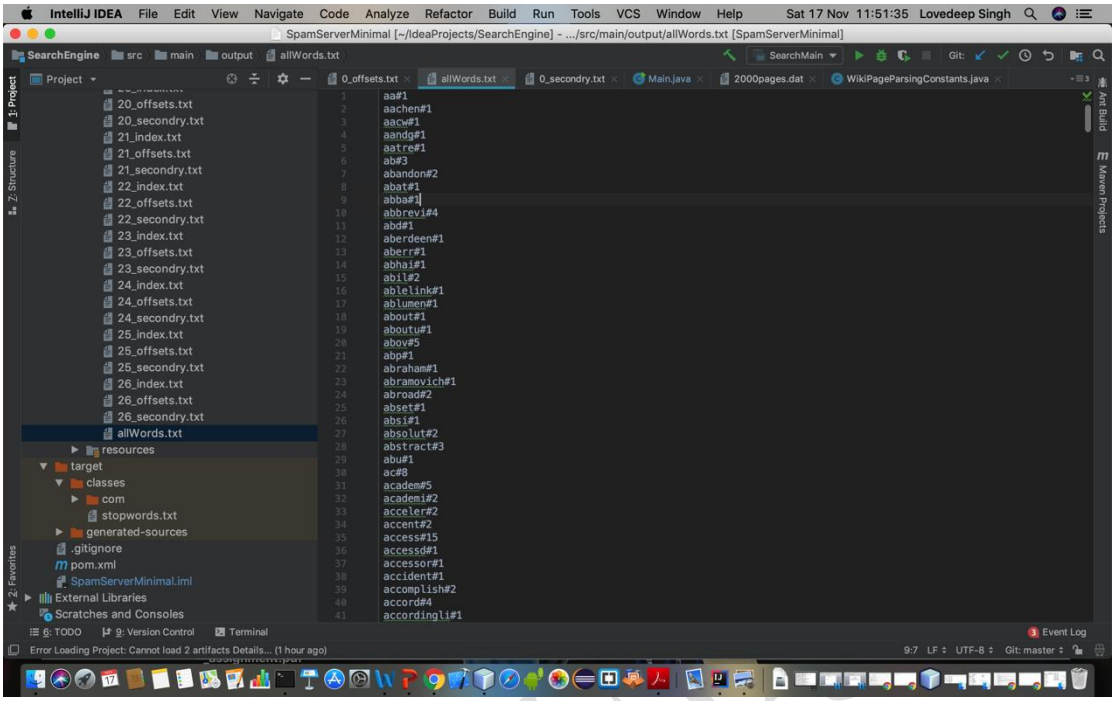

Figure 12

0_secondary.txt

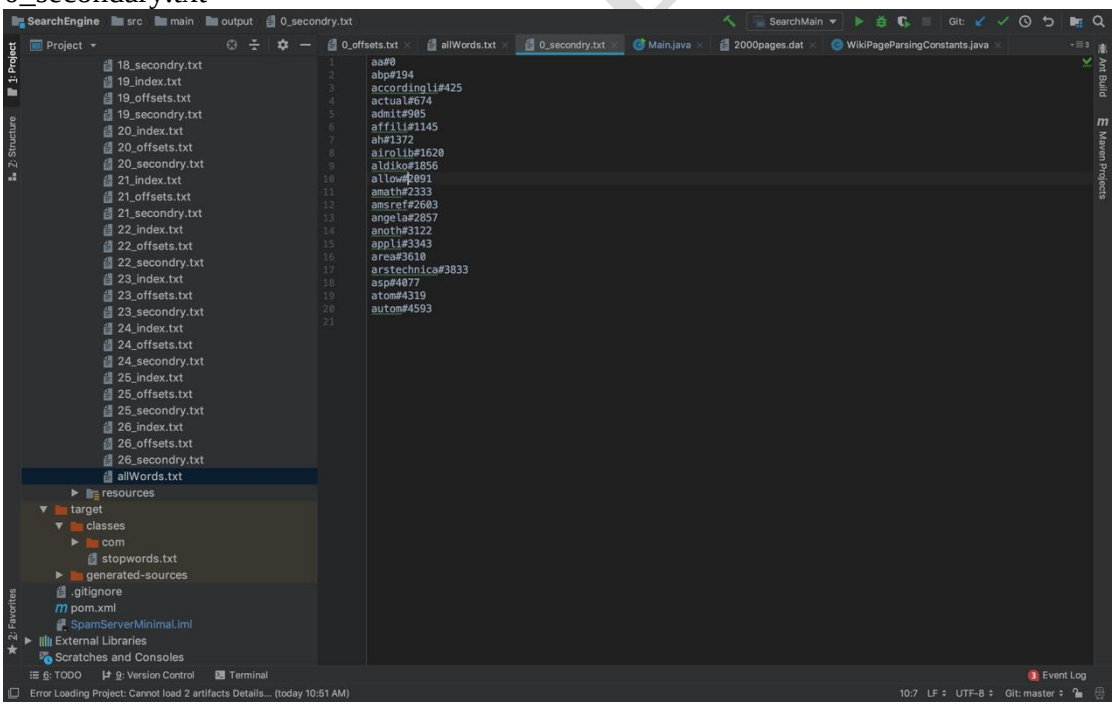

Figure 13

0_offsets.txt

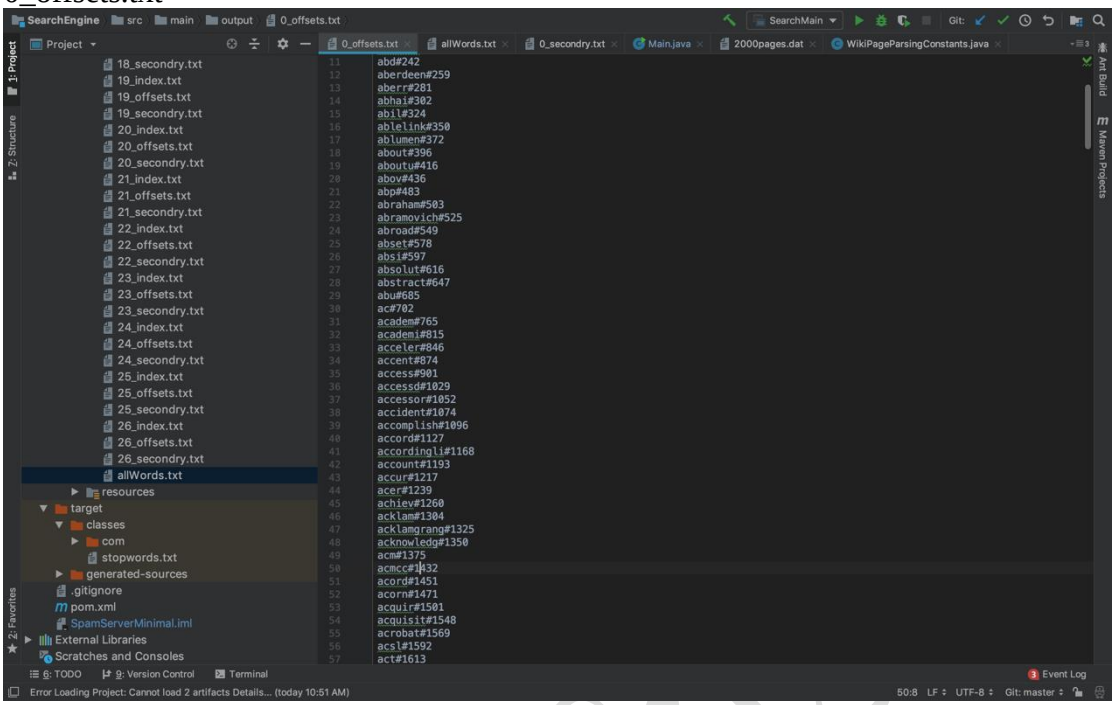

Figure 14

0_index.txt

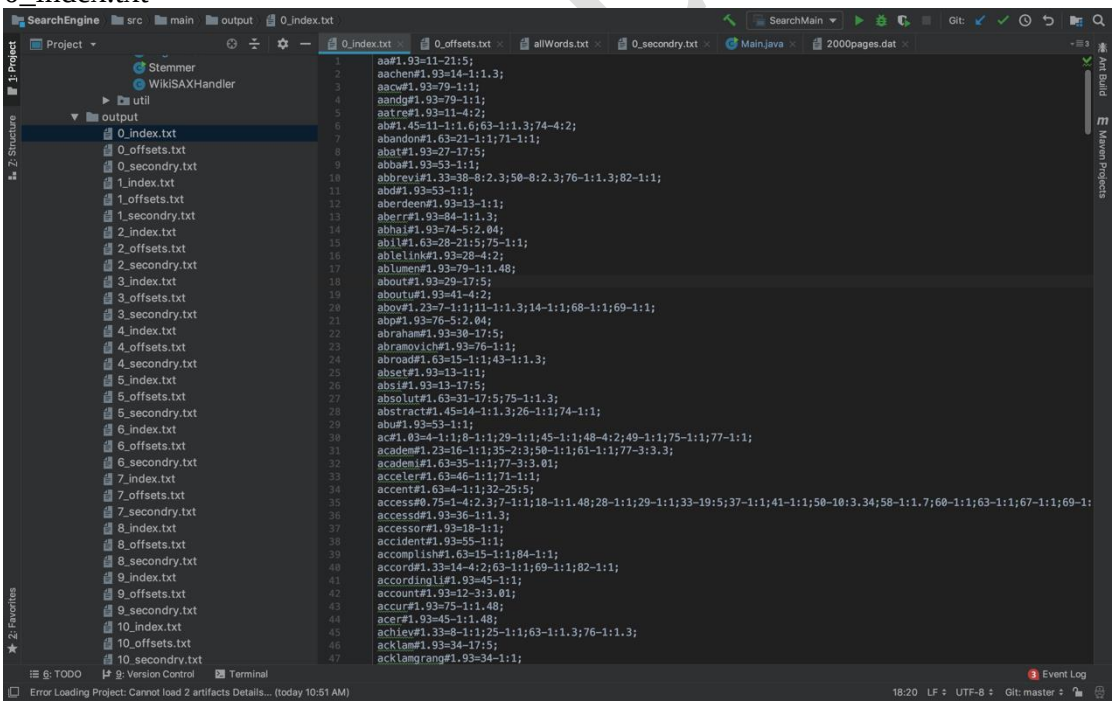

Figure 15

SEARCH

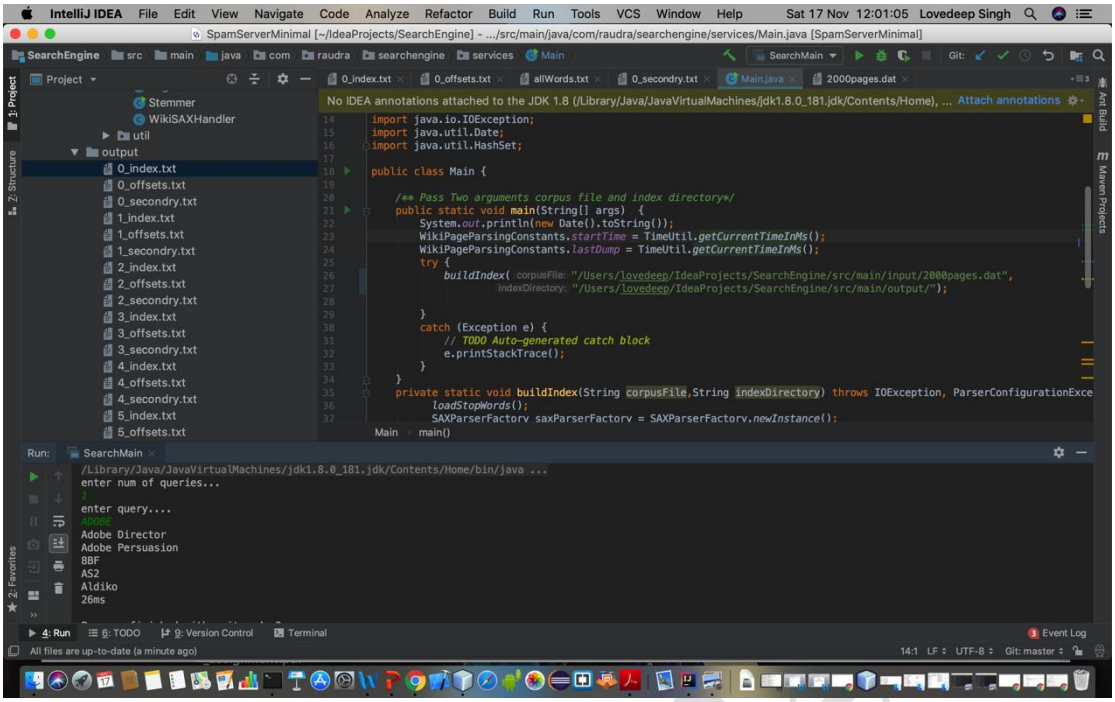

Figure 16

CHAPTER 5
RESULTS AND DISCUSSION

With the able guidance of our supervisor and dedicated effort put in by us, we are able to complete the project in time. The source code of the project can be found at the following link:

https://github.com/krishna0545/SearchEngine

LEARNINGS

Throughout the course of the project, we touched a number of aspects both in technological and cooperative work domain and learnt valuable lessons.

TECHNOLOGY STACK**:**

- XML
- JAVA LANGUAGE
- INTELLIJIDEA

CONCEPTS IN SEARCH ENGINE WORLD**:**

- Tokenization, stemming, parsing, case folding, inverted indexes, stop words, inverted document frequency, etc.

MORAL PRINCIPLES**:**

- Teamwork, discipline, punctuality, etc.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

Building a TEXT SEARCH ENGINE is feasible. However, it is not easy to build one search engine better than existing one. The main problem lies in during analyzing a document, which parts to consider and which not for indexing. As we go more towards full text search from exact value search it becomes more and more challenging. The father of all search engines GOOGLE has got the best recipe of algorithms based on machine learning, which makes it the most popular search engine today.

Text Search Engine is a basic prototype of a search engine. The name is actually misnomer; it is more of a sorting engine than a searching one, just to reply whether the word is present in documents or not is limited to searching, but to retrieve top relevant documents is the aspect of sorting. The basic functions provided are words extracting, creating inverted index, searching.

A number of enhancements are possible in this project in future.

Static enhancements:

- Increase in indexed database
- Inclusion of better stemming algorithms
- Redefinition of stopwords

Dynamic enhancements:

- Spell check
- Extra search functionalities

References:

Wikipedia:

https://en.wikipedia.org/wiki/XML

https://en.wikipedia.org/wiki/Parsing

https://en.wikipedia.org/wiki/Web_search_engine

Webopedia:

https://www.webopedia.com/TERM/S/search_engine.html

Wikibooks:

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Parsing

Technopedia:

https://www.techopedia.com/definition/3854/parser

New idea engineering:

http://www.ideaeng.com/database-full-text-search-0201