DECLARATION

We hereby declare that the work which is being presented in this project report entitled **"Search Engine",** in partial fulfillment of the requirement for the award of the degree of **MASTER OF COMPUTER APPLICATIONS** submitted to Department of Computer Applications, National Institute of Technology, Kurukshetra is an authentic work done by us during a period from 2015 to 2017 under the Guidance of Dr. **Kapil Gupta.**

The work presented in this project report has not been submitted by us for the award of any other degree of this or any other Institute/University.

Signature

Gabbar Singh Sisodiya

Roll no. 5141080

Signature

Ratnesh Singh

Roll no. 5142033

Signature

Brajendra

Roll no. 5141056

This is to certify that the above statement made by the candidate is correct to best of my knowledge and belief.

Date :

Place :

Signature

Name of the Supervisor

ACKNOWLEDGEMENT

We would like to sincerely thank the National Institute of Technology Kurukshetra for giving us this opportunity of taking up such a challenging project which has enhanced our knowledge about the Information Retrieval and to make a project on Search Engine and how to make it easily available to users and clients of the agency.

We are very grateful to Dr Kapil Gupta Project Supervisor, under whose guidance and assistance we were able to successfully complete our project.

Last but not the least; we also thank the below-mentioned honorable dignitaries and task-masters who have played a major role in our project to the sky of glory. This is a special thanks to them for sparing their precious time, fitting my out-of-the-way appointent into their diary and giving almost all the information required by us in an unbelievably amicable manner.

Without the priceless contribution and coveted guidance of all the above-mentioned people, this project would have never got a shape of reality and emerged before all of you in the manner and in the style as it now appears.

ABSTRACT

The aim of this project is about measuring the effectiveness of standard Information Retrieval systems. The standard approach to information retrieval system evaluation revolves around the notion of relevant and non-relevant documents. Basic measures for information retrieval effectiveness that are standardly used for document retrieval are Precision and Recall. So using standard information retrieval systems we define specific queries and then answer these specific queries. Measure precision and recall values with the standard information retrieval systems.

LIST OF FIGURES

LIST OF TABLES

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

There are many Information retrieval systems. Information retrieval has developed as a highly empirical discipline, requiring careful and thorough evaluation to demonstrate the superior performance of different novel techniques used by IR systems on representative document collections.

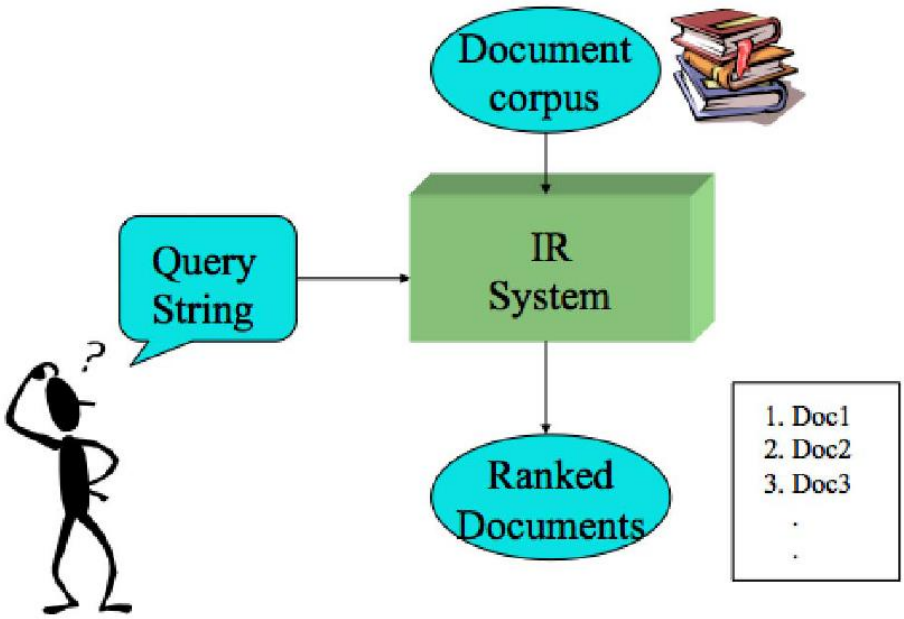

Fig 1: Typical Information Retrieval task

The group of documents over which information retrieval is performed is called as collection. It is also called as Document corpus. User submits query. Query is what the user conveys to the computer in an attempt to communicate the information need. Information need is the topic about which the user desires to know more. Documents retrieved by IR systems are arranged according to their relevance to a given search query. A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need.

There exist many IR systems like Google, Bing, MSN and many more but the performance and quality are not known. The main aim of this work is to measure the effectiveness of search engines.

1.2 Standard information retrieval systems:

The great explosion of Internet and electronic data repositories has brought lots of data in our reach. Day by day data is increasing exponentially, not only on Internet but also on desktop computers. It is very impractical to look for a picture or an audio file out of many folders and subfolders on a desktop. Although user can classify data, crawling through many categories and subcategories of data, which in not an efficient way. There is a need for alternate and dynamic ways of finding information. That's where information retrieval systems come into existence.

Information retrieval system finds material of an unstructured nature that satisfies and information need of user from within large collections.

1.3 Queries

User defines queries to IR systems. Query is what the user conveys to the computer in an attempt to communicate the information need. Queries have an influence on matching result. Queries must match with content description of multimedia documents.

1.4 Search Engine

A search engine is simply a database of web pages, a method for discovering and storing information about new web pages, and a way to search that database. Therefore, in the simplest form, search engines perform three basic tasks: 1. Traverse the web and determine important words (Crawling). 2. Build and maintain and index of sites keywords and links (indexing) 3. Present search results based on reputation and relevance to users keyword combinations (searching).

CHAPTER 2

LITERATURE SURVEY

Linked data source are efficient way to store web data with use of RDF, rather RDBMS and Keyword relationship graph at set and element level are to be searched. Finally multi-interrelationship between different elements is used to find out tanking order of the retrieved documents. Touting keywords only to relevant sources can reduce the high cost of searching for structured results that span multiple sources. The routing plans, produced can be used to compute results from multiple sources.

Space search ordering algorithm is take to perform the aggregation on the data set and finds out relevance score for the each keyword in entered search query. This is similar to OLAP analysis on data cube, searches the relevant k cells on cube by the space search ordering approach. Here, RDF is modeled as undirected data graph and Minimum keyword tree generation method is used to reduce the searching complexity. Searching and retrieval of data item is done from generated minimum keyword tree and at the end to put ranking on result score function is use. Ranking keyword search results with Collective Importance Ranking called as CI-Rank. Random walk with message passing (RWMP) approach is used to find out the cohesiveness in the result tree of data retrieved. CI-Rank will help to find out the better ranking, as it cans be later useful for finding precision of the pages to calculate means reciprocal rank, as it helps for relevance of web pages. For the RDBMS search area, CI rank is used to check the performance of search engine basically these databases are normalized to remove redundancy and compare normalization used for wed data. To find general approach to study learning objectives via search engines like SCORM and CORDRA which are different form the general purpose search engine as Google. And also to compare ranking order this add use and generalized structure for SERPs Search engine Retrieved Page Links.

Referred Paper	Purpose	Conclusion
Keyword Query Routing IEEE Transactions on Knowledge and Data Engineering, 2014	Routing Plans preparation methods are provided at element and set level graphs of web data.	If no any semantic between keywords is found methods will not be applied.
Efficient Keyword based search for Top K-cells in test cube IEEE Transaction vol 23, no 12, DEC 2011.	It gives idea about keyword search on DB for top k relevant result similar working as RDBMS	This is not applied for all the database schemas except RDBMS structure.
Paper 2: Keyword Proximity search over large and compel RDF database 2012 IEEE Transaction.	To find search procedure on RDF database reduces searching complexity.	Dynamic search is adapted for RDF databases.

Table1. Summary of Literature Review

CHAPTER 3

LUCENE

3.1 About Lucene

Apache Lucene is a full featured, high performance, scalable text Search engine library. It is written in java. Lucene has been recognized for its utility in implementation of Internet search engine. It provides full text indexing and searching. It was originally written by Doug Cutting. It belongs to Apache Jakarta family of projects.

Lucene is not a complete application and so it is not ready to use application like web search engine, rather it is code library that can be easily integrated into any application. It is popular because of its simplicity. User need not know how it does the indexing and searching, rather just have to learn few classes of Lucene library for implementation.

There are many applications, which uses Lucene. For example

- Eclipse Lucene: Eclipse IDE uses for searching its documentation
- Nutch: open source web search engine
- Twitter Trends: Twitter analyzing tool
- 7digital: Digital media delivery Company
- Linked In, Apple, IBM and many more

Lucene can be seen as layer above which sits the application as shown in fig 2

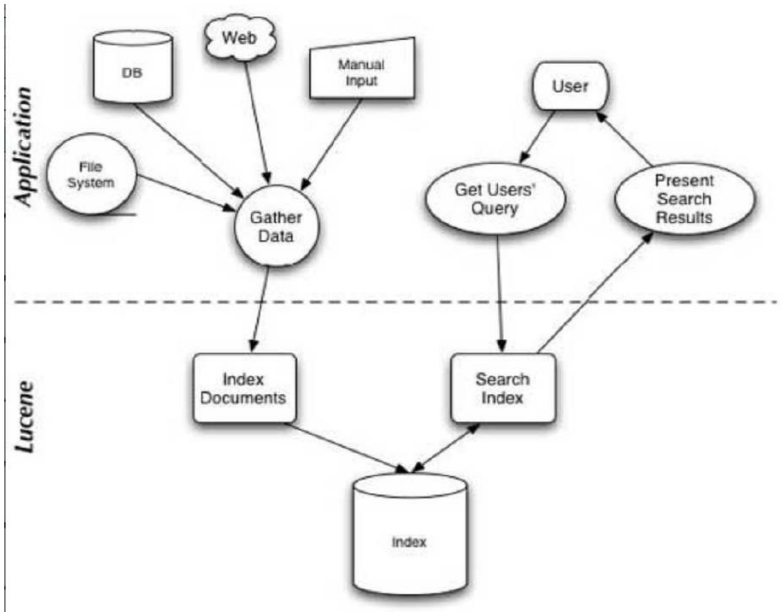

Fig 2: A typical application integration with Lucene

As seen in above figure Lucene is not only used on web, but also on local File system, databases, manual input and many other sources of data. It can index and search any data that can be converted into text. It works on all data formats like html, pdf, txt, and Microsoft word. It gathers the data and indexes those documents. And later when user gives a query, it performs a search on index generated.

Lucene offers many features. It can provide ranked searching which means best results are returned first .It also provides fielded searching, i.e. user can search for content in title field, author or content field. It supports different types of queries like phrase queries, range queries, wildcard queries. Lucene also supports simultaneous searching and updating. It is highly flexible and scalable for any number of documents. It also does sorting, filtering, highlighting search results.

3.2 Lucene Indexing

Indexing is the center concept of any search engine. Indexing is the process of converting original data into an efficient lookup, which helps for rapid searching.

Suppose if we want to look for a file with a specific word, then we could have a program that sequentially scans for all the files and look for a file with a specific word. But this is quite impractical when the file set is large. Here comes the importance of indexing. In these cases, first the text must be indexed into a format, which helps for rapid searching and this process eliminates the slow scanning process and it is called indexing. And the output of it is called index. Index is a data structure, which facilitates rapid search for the words present in it.

Lucene converts any format of data to text and then indexes it. Lucene uses different parsers for different documents like HTML parser for html documents .HTML parser does some preprocessing by filtering html tags and so on. The parser outputs text content. Lucene Analyzer extracts the tokens and other related information and stores it in index files. Like html there are different parsers for pdf, Microsoft word and text files. Below figure shows the indexing architecture of Lucene.

3.2.1 Indexing example using Lucene:

```
Public static void createIndex() throws

CorruptIndexException, LockObtainFailedException,

IoExcetption {

        Analyzer analyzer = new StandardAnalyzer();

        Boolean recreateIndexIfExists = ture;
```

```
        IndexWriter indexWriter =new IndexWriter (INDEX_DIRECTORY, analyzer,
recreateIndexIfExists);

        File dir = new File (FILES_TO_INDEX_DIRECTORY);

        File[] files = direlistFiles();

        For (File file : files ) {

                Document document = new Document();

                String path = file.getCanonicalPath();

                Document.add(new Field(FIELD_PATH, path, Field.Store.YES,
Field.Index.UN_TOKENIZED) );

                Reader reader = new FileReader(file);

                Document. Add(new field(FIELD-CONTENTS, reader) );

                indexWriter.addDocument(document);

        }

        indexWriter.optimize();

        indexWriter.close();

}
```

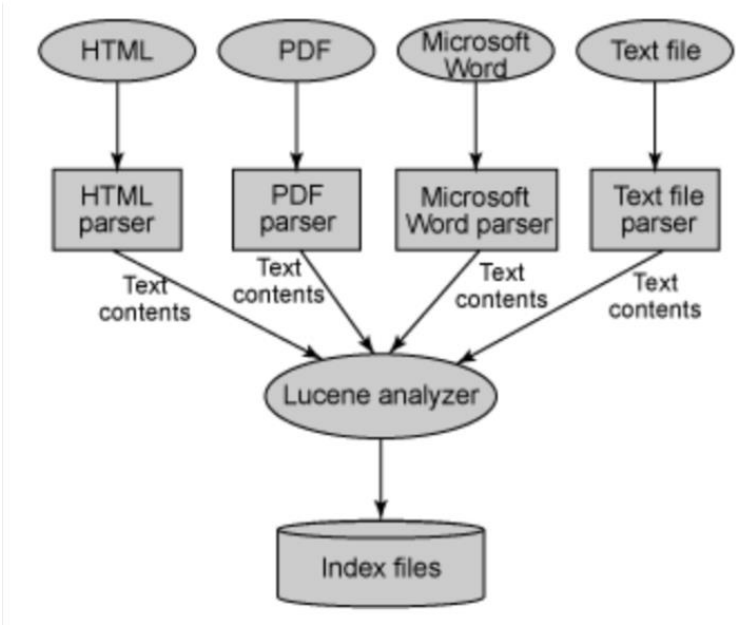

Fig 3: Indexing architecture of Lucene

Indexer program needs two important command line arguments.

- A path to a directory where Lucene index is to be stored
- A path to a directory which contains files to be indexed

Running the above Indexer program will create a Lucene index. Indexer prints the names of files it indexes. It displays the total number of documents indexed and also time took in milliseconds. This includes time needed for directory traversal and time needed for indexing (See Appendix A1).

3.2.2 Indexing classes of Lucene:

Here are some indexing classes that are used for indexing process using lucene.

- Index Writer
- Directory
- Analyzer
- Field

IndexWriter:

It is a central component of indexing process. It creates the index and also adds documents to the existing index.

It just gives the write access to the index, but does not allow reading or searching on index.

Directory:

Directory class represents the location of Lucene index. When indexing is to be done, then Directory class is given to the IndexWriter so that IndexWriter creates an index in a location specified by Directory class. There are different implementations of Directory like FSDirectory, RAMDirectory. Both have similar interfaces. FSDirectory stores the index on a disk. But RAMDirectory holds all its data in memory. It can be destroyed after application terminates. Thus, searching on index generated by RAMDirectory is faster than index generated by FSDirectory, because fetching from hard disk is bit more slow than from memory.

Analyzer:

IndexWriter specifies analyzer. And this analyzer is responsible for extracting tokens from the text. There are different implementations of analyzers. Few analyzers skips stop words like the, is, at. Few other analyzers index words with case insensitivity. Depending on the requirement of application, suitable analyzer can be used.

Below table shows different analyzers:

Analyzers	Description
Standard Analyzer	A sophisticated general-purpose analyzer.
Whitespace Analyzer	A very simple analyzer that just separates tokens using white space.
Stop Analyzer	Removes common English words that are not usually useful for indexing.
Snowball Analyzer	An interesting experimental analyzer that works on word roots (a search on *rain* should also return entries with *raining*, *rained*, and so on).

Table 2: Lucene Analyzers

Other than the above-mentioned analyzers in table, there are also language specific analyzers for German, French, Russian and others.

Example:

```
IndexWriter IndexWriter =
        new IndexWriter ("index-directory", new StandardAnalyzer (), true);
```

This example uses all the 3 above-mentioned classes. First parameter for Index Writer is “index-directory”, this is the location where index has to be created. Second parameter tells which document analyzer should be used. In this case it uses StandardAnalyzer.

Document:

Document is a bundle of data or collection of fields like title, author, content and so on. For every file that is to be indexed, a Document class is created, populated with fields and added to the index. Document can be simple text file, webpage, email or a message.

Example:

```
Document doc = new Document ();
doc.add (new Field ("description", Field.Store.YES, Field.Index.TOKENIZED));
```

Field:

Document consists of one or more fields. Upon these fields in index a search can be done. There are different types of fields.

- Keyword: These are not analyzed but indexed, so that the original value is preserved
- Unindexed: These fields are not analyzed or indexed, but simply stored, so that it can retrieved as the way they are during a search.
- Unstored: These are analyzed and indexed but not stored.
- Text: This is analyzed and indexed.

3.3 Lucene Searching

Searching is a process of looking words in an index in order to find out in which documents they appear in file set. The search term can be a single word, phrase query wildcard query etc.

The quality of search is measured using Recall and Precision. Recall tells how efficiently are relevant documents retrieved and precision tells how efficiently are the irrelevant documents filtered.

3.3.1 Searching example using Lucene:

```
public static void searchIndex(String searchString) throws IOException, ParseException
{
        System.out.println("Searching for '" + searchString + "'");
        Directory directory =  FSDirectory.getDirectory(INDEX_DIRECTORY);
        IndexReader indexReader = IndexReader.open(directory);
        IndexSearcher indexSearcher = new IndexSearcher(indexReader);
```

```
        Analyzer analyzer = new StandardAnalyzer();
        QueryParser queryParser = new QueryParser(FIELD_CONTENTS, analyzer);
        Query query = queryParser.parse(searchString);
        Hits hits = indexSearcher.search(query);
        System.out.println("Number of hits: " + hits.length());
        Iterator<Hit> it = hits.iterator();
        while (it.hasNext())
        {
                Hit hit = it.next(); Document document =    hit.getDocument();
                String path = document.get(FIELD_PATH);
                 System.out.println("Hit: " + path);
        }
}
```

Searcher program above needs two important command line arguments.

- A path to the index created with Indexer
- A query to use to search the index

Searcher program returns the documents that match the query in the form of Hits. It also prints number of documents matched. For performance reasons not all the hits are returned. Only few are printed (See Appendix A2).

3.3.2 Searching classes of Lucene:

- IndexSearcher
- Term
- Query
- TermQuery
- Hits
-

IndexSearcher:

IndexSearcher is the main link to index. It opens the index in the read only mode. It contains many search methods. Few of those are implemented in its parent class Searcher. IndexSearcher takes query object as a parameter and returns Hits object.

Example:
IndexSearcher is= new IndexSearcher (FSDirectory.getDirectory
(“/volumes/User/project/index”, false));

Term:

Term is the basic unit of searching. It consists of pair of string elements. They are name of field and value of field. Term objects are together used with TermQuery while searching.

Example:

```
Query q=new TermQuery (new Term (“title”, “Manning”));
Hits hits=is.search (q);
```

Query:

There are different Query classes like PhraseQuery, BooleanQuery and few others. Query class is the parent of all the above classes.

TermQuery:

TermQuery is Lucene’s basic query type. It is used to search a field with a specific value. It is mostly used together with Term.

Hits:

Hits are a set of documents that match the query.

CHAPTER 4

LUCENE INDEX STRUCTURE

Lucene stores its data in the form of inverted index. An inverted index is an inside-out arrangement of documents in which terms take center stage. Each term points to a list of documents that contain it. It is an index data structure mapping terms and the documents that contain it. The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added.

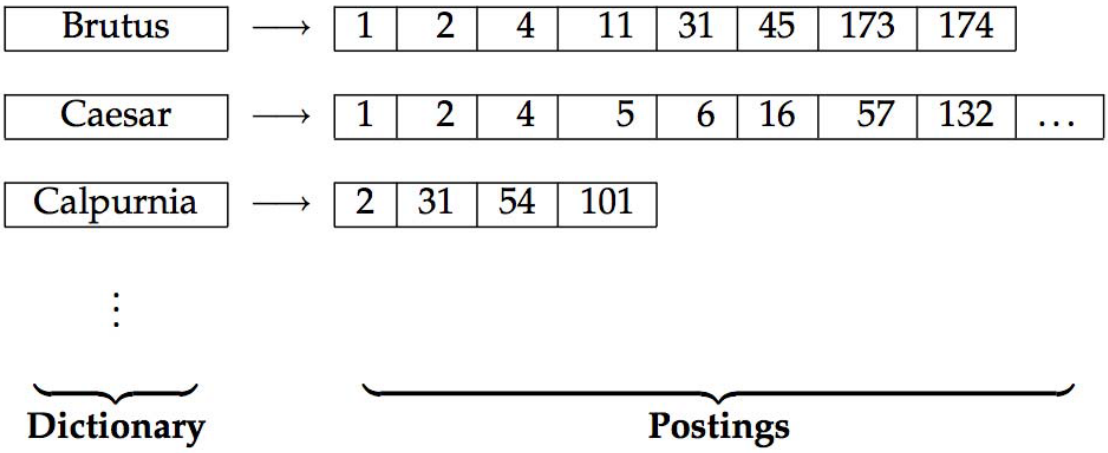

Fig 4: Inverted index

Example:

The inverted index as seen above contains two parts - Dictionary and Postings. Dictionary contains the terms and Postings contains the list of documents that contain the term.

Lucene index structure is shown in fig 5.

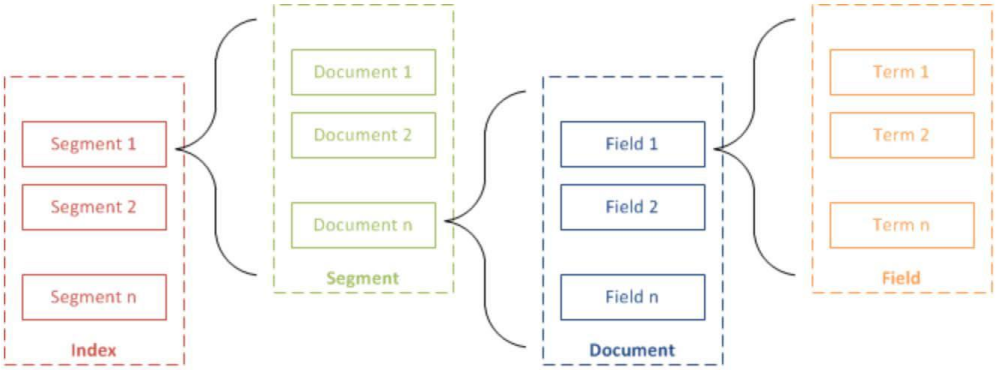

Fig 5: Lucene index structure

Segments:

Lucene index consists of sub-indexes or segments. These are independent index, and these can be searched separately.
Index is created by:

- Creating segments for newly added documents
- Merging existing segments

Each segment contains the following:

- Field names: Contains set of field names used in the index.
- Stored Field values: Contains attribute-value pairs for each document. Attribute is the field name.
- Term dictionary: Contains all the terms in all of the indexed fields of all the documents.
- Term Frequency data: Contain for each term in the dictionary, number of all documents that contain the term, and also number of times it occurs in each document.
- Term Proximity data: For each term in the document, it contains the position of terms in the documents.
- Term Vectors: Contains the term text and term frequency.
- Deleted documents: Indicates which files are deleted. This is optional.

The number of documents to be indexed and the number of documents a segment can contain determine the number of segments. Below figure indicates this.

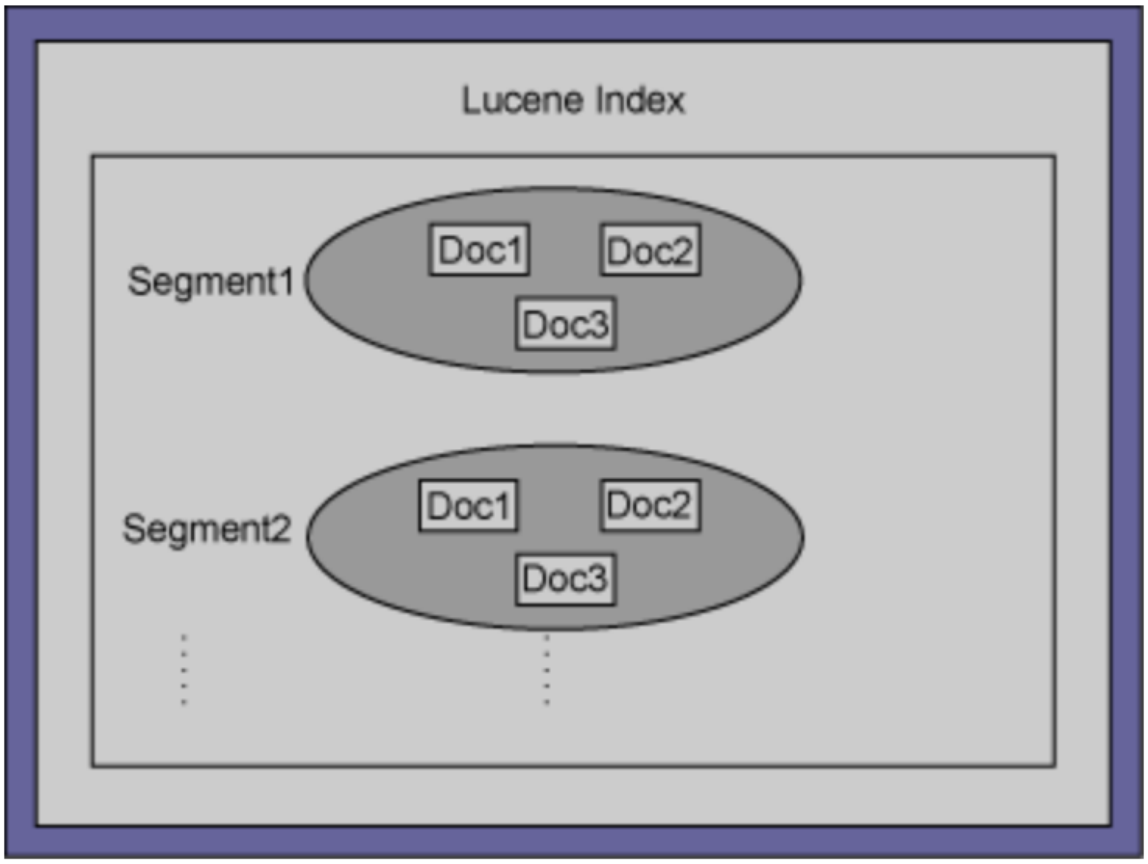

Fig 6: Logical view of index files

Document Numbers

Internally Lucene refers to documents with an integer document number. The first document added to index is numbered zero and subsequent documents are numbered one more than previous.

File Naming

Index contains different files with different extensions with different information. Below table summarizes different files.

Name	Extension	Description
Segments Files	Segments.gen, segments_N	Stores information about segments
Lock File	Write.lock	The write lock prevents multiple Index Writers from writing to the same file.
Compound File	.cfs	An optional “virtual” file consisting of all the other index files for systems that frequently run out of file handles.
Fields	.fnm	Stores information about the fields
Field Index	.fdx	Contains pointers to field data
Field Data	.fdt	The stored fields for documents
Term Infos	.tis	Part of the term dictionary, stores term info
Term Info Index	.tii	The index into the Term Infos file
Frequencies	.frq	Contains the list of docs which contain each term along with frequency
Positions	.prx	Stores position information about where a term occurs in the index
Norms	.nrm	Stores offset into the document data file
Term Vector Index	.tvx	Stores offset into the document data file
Term Vector	.tvd	Contains information about each document that has term vectors.
Term Vector Fields	.tvf	The field level info about term vectors
Deleted Documents	.del	Info about what files are deleted

Table 3: Names and extensions of the files in Lucene

Structure of few of the above files is shown below.

Fields information file

Column name	Data type	Description
FieldsCount	VInt	The number of fields.
FieldsName	String	The name of one field.
FieldBits	Byte	Contains vrious flags. For example, if the lowest bit is 1, it means this is an indexed field; if 0, it's a nonindexed field.

Table 4. Structure of fields information file

Frequency File

Column name	Data type	Description
DocDelta	VInt	It determines both the document number and term frequency. If the value is odd, the term frequency is 1; otherwise, the freq column determines the term frequency.
Freq	VInt	If the value of DocDelta is even, this column determines the term frequency.

Table 5. Structure of the frequency file

Position file

Column name	Data type	Description
PositionDelta	VInt	The position at which each term occurs within the documents.

Table 6. Structure of the position file

CHAPTER 5

RESULTS

5.1 Serching

Searching is a process of looking words in an index in order to find out in which documents they appear in file set. The search term can be a single word, phrase query wildcard query etc.

The quality of search is measured using Recall and Precision. Recall tells how efficiently are relevant documents retrieved and precision tells how efficiently are the irrelevant documents filtered.

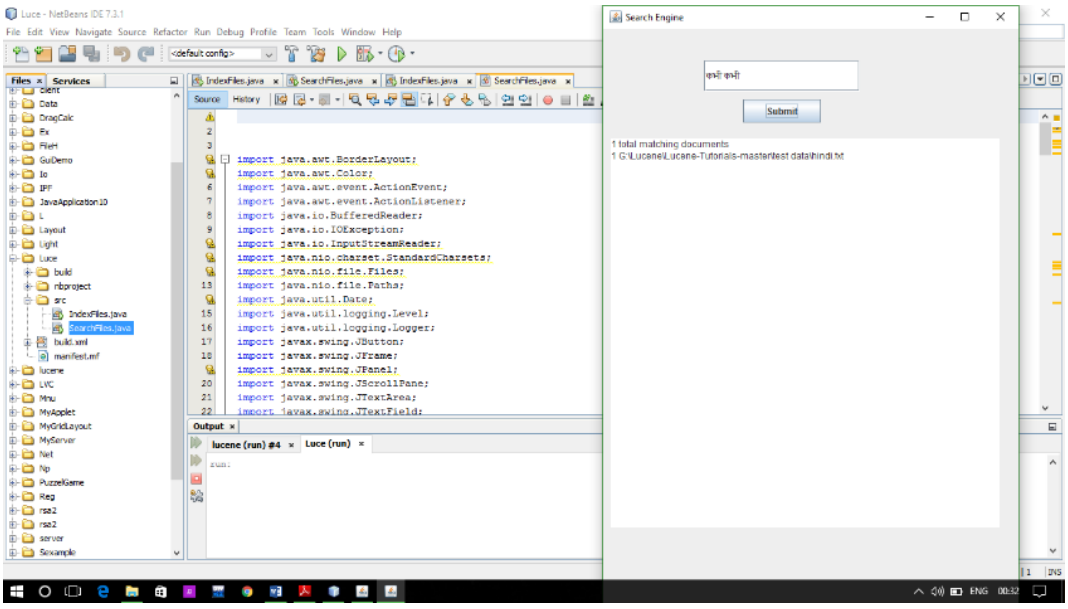

Fig 7 Query and Result

In above image we give query “Kabhi kabhi” in hindi language and it’s showing result of one file with it’s path, which contain “Kabhi kabhi”, It’s also wotks for multilingual language,

CONCLUSION

The aim of the project was to measure the effectiveness of standard information retrieval systems using specific queries. So using the Lucene, an information retrieval library, Corpus of multimedia documents, and few specific queries, the experiment is conducted.

Precision and recall are the measures for measuring the effectiveness of any information retrieval systems. These two measures are based on relevant and retrieved set of documents. Looking at high values of precision it is clear that, most of the returned results are relevant to the query. Which means less number of false positives, proving Lucene is efficient in a way.

Precision can be seen as a measure of exactness or quality, whereas recall is a measure of completeness or quantity. In simple terms, high recall means that an algorithm returned most of the relevant results. But in our results the recall values are not high, that means not all relevant documents are returned. While high precision means that an algorithm returned substantially more relevant results than irrelevant.

BIBLIOGRAPHY

1. Lashkari, Arash Habibi, Fereshteh Mahdavi, and Vahid Ghomi. "A boolean model in information retrieval for search engines." *Information Management and Engineering, 2009. ICIME'09. International Conference on*. IEEE, 2009.
2. Lee, Uichin, Zhenyu Liu, and Junghoo Cho. "Automatic identification of user goals in web search." *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005.
3. Khan, Junaid. "Comparative study of information retrieval models used in search engine." *Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on*. IEEE, 2014.
4. Liu, Yiliang, and Guishi Deng. "Domain-oriented retrieval model research based on meta-search." *Service Operations and Logistics, and Informatics, 2008. IEEE/SOLI 2008. IEEE International Conference on*. Vol. 1. IEEE, 2008. Sujatha, Pothula, and P. Dhavachelvan. "A Review on the Cross and Multilingual Information Retrieval." *International Journal of Web & Semantic Technology* 2.4 (2011): 115.
5. Sharma, Deepika. "Stemming algorithms: A comparative study and their analysis." *International Journal of Applied Information Systems* 4.3 (2012): 7-12.
6. Kobayashi, Mei, and Koichi Takeda. "Information retrieval on the web." *ACM Computing Surveys (CSUR)* 32.2 (2000): 144-173.
7. Belyaeva, Evgenia, et al. "Using semantic data to improve cross-lingual linking of article clusters." *Web Semantics: Science, Services and Agents on the World Wide Web* (2015).
8. Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze- An introduction to information retrieval, Cambridge university, England.
9. Gospodnetic, Otis; Eric Hatcher, Michael McCandless (28 June 2009). Lucene in Action (2^{nd} Ed.). Manning Publications.
10. http://wiki.apache.org/lucene-java/PoweredBy
11. http://oak.cs.ucla.edu/cs144/projects/lucene/
12. Lucene.apache.org
13. http://www.ibm.com
14. http://nlp.stanford.edu/IR-book/
15. Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008

APPENDIX

A1 Creating Index (Code)

```
package lucene;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.LongField;
import org.apache.lucene.document.StringField;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig.OpenMode;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
```

```
import java.util.Date;
public class IndexFiles {
  private IndexFiles() {}

  /** Index all text files under a directory. */
 public static void main(String[] args) {

    String indexPath = "G:\\Lucene\\Lucene-Tutorials-master\\index";
    String docsPath = "G:\\Lucene\\Lucene-Tutorials-master\\test data";
    boolean create = true;
    final Path docDir = Paths.get(docsPath);
    if (!Files.isReadable(docDir)) {
      System.out.println("Document directory.. '" +docDir.toAbsolutePath()+ "' does not exist or is not readable, please check the path");
      System.exit(1);
    }
    Date start = new Date();
    try {
      System.out.println("Indexing to directory '" + indexPath + "'...");
      Directory dir = FSDirectory.open(Paths.get(indexPath));
      Analyzer analyzer = new StandardAnalyzer();
      IndexWriterConfig iwc = new IndexWriterConfig(analyzer);
      if (create) {
        // Create a new index in the directory, removing any
        // previously indexed documents:
        iwc.setOpenMode(OpenMode.CREATE);
      } else {
        // Add new documents to an existing index:
        iwc.setOpenMode(OpenMode.CREATE_OR_APPEND);
```

```
    }

    // iwc.setRAMBufferSizeMB(256.0);

    IndexWriter writer = new IndexWriter(dir, iwc);
    indexDocs(writer, docDir);

    writer.close();
    Date end = new Date();
    System.out.println(end.getTime() - start.getTime() + " total milliseconds");
  } catch (IOException e) {
    System.out.println(" caught a " + e.getClass() +
     "\n with message: " + e.getMessage());
  }
 }

 static void indexDocs(final IndexWriter writer, Path path) throws IOException {
  if (Files.isDirectory(path)) {
    Files.walkFileTree(path, new SimpleFileVisitor<Path>() {
      @Override
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
        try {
          indexDoc(writer, file, attrs.lastModifiedTime().toMillis());
        } catch (IOException ignore) {
          // don't index files that can't be read.
        }
        return FileVisitResult.CONTINUE;
      }
```

```
      });
    } else {
      indexDoc(writer, path, Files.getLastModifiedTime(path).toMillis());
    }
  }


  /** Indexes a single document */
  static void indexDoc(IndexWriter writer, Path file, long lastModified) throws
IOException {
    try (InputStream stream = Files.newInputStream(file)) {
      // make a new, empty document
      Document doc = new Document();
      Field pathField = new StringField("path", file.toString(), Field.Store.YES);
      doc.add(pathField);
      doc.add(new LongField("modified", lastModified, Field.Store.NO));
      doc.add(new TextField("contents", new BufferedReader(new
InputStreamReader(stream, StandardCharsets.UTF_8))));


      if (writer.getConfig().getOpenMode() == OpenMode.CREATE) {
        // New index, so we just add the document (no old document can be there):
        System.out.println("adding " + file);
        writer.addDocument(doc);
      } else {
        System.out.println("updating " + file);
        writer.updateDocument(new Term("path", file.toString()), doc);
      }
    }
  }
}
```

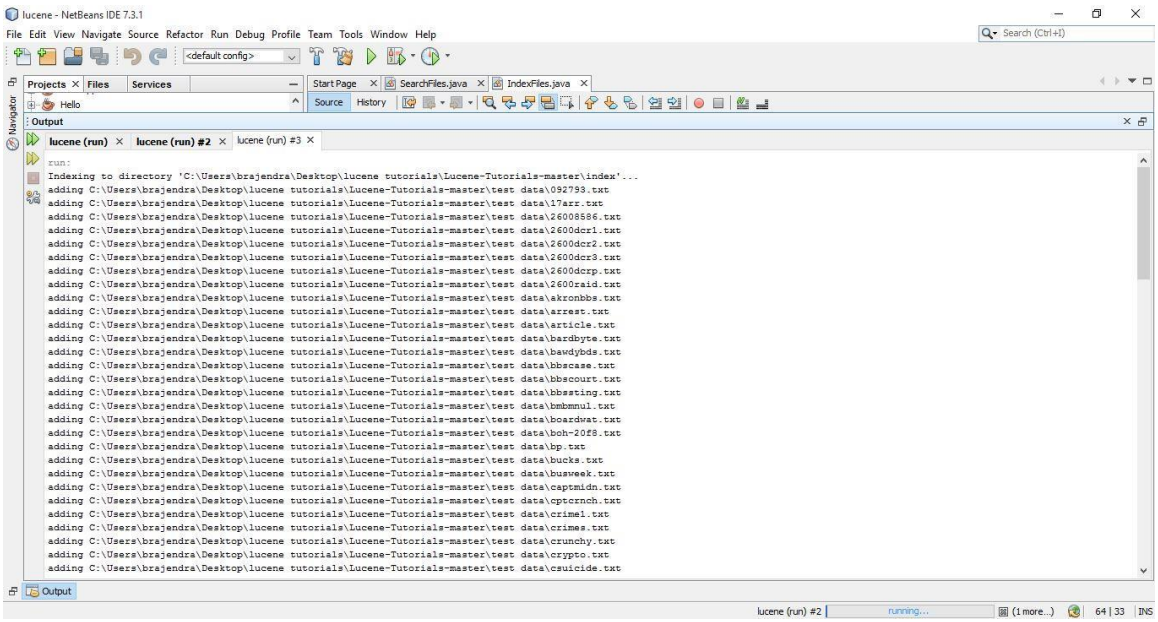

Figure 8: Run Index.java

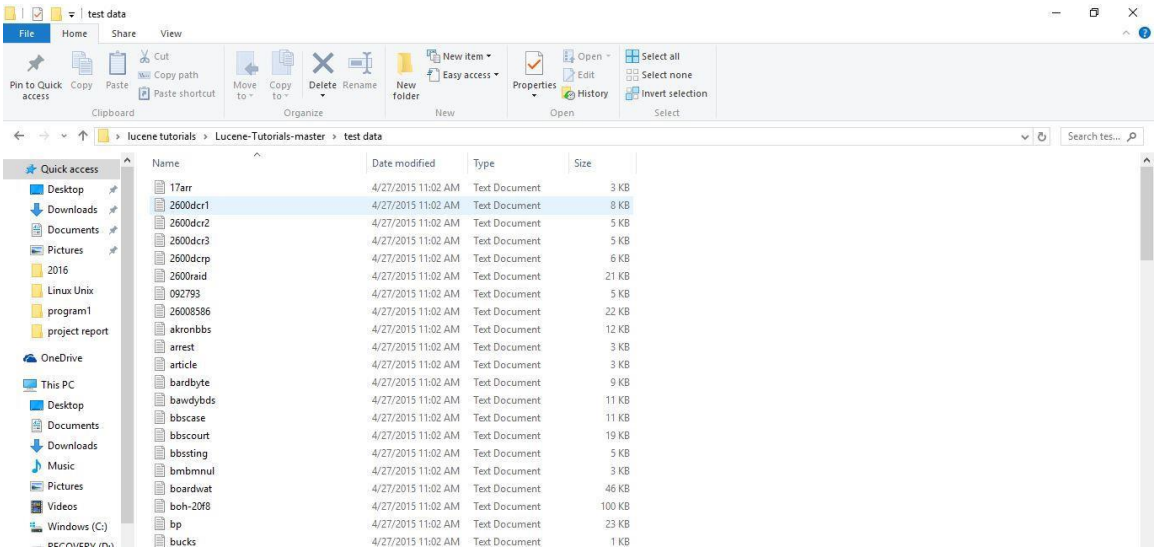

Figure *9: Corpus (*Simple data file which is content news article)

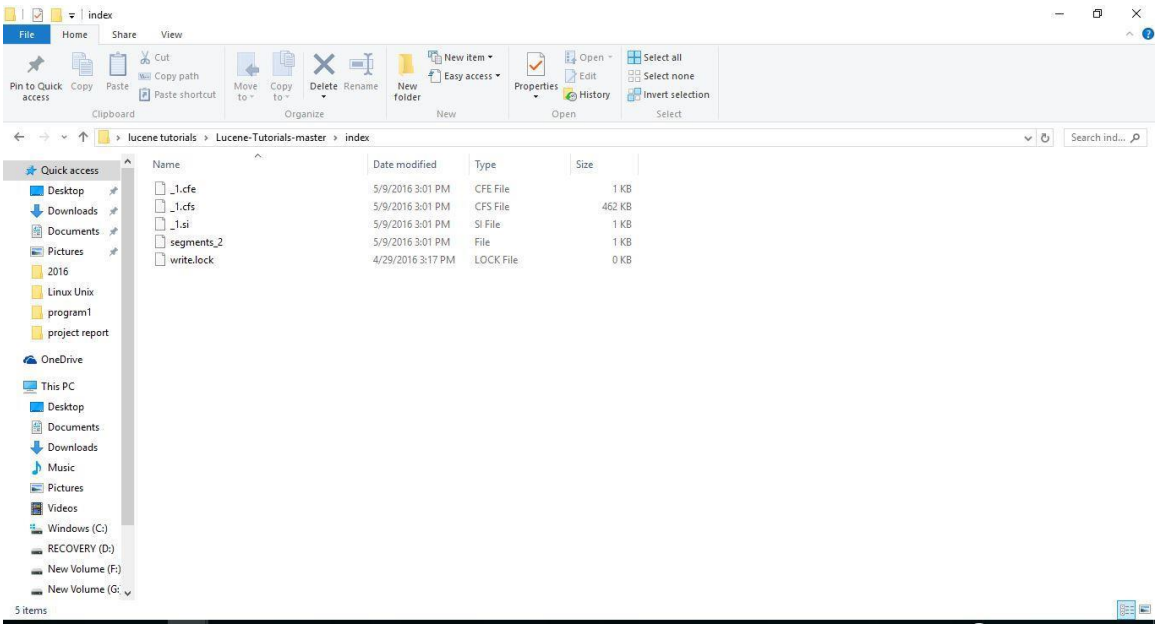

Figure 10: Index (After running index.java class)

A2 Searching (Code)

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.FSDirectory;

/**
 * Simple GUI based search demo.
 */
public class SearchFiles extends JFrame implements ActionListener {

    JTextField t;
    JButton b;
    JTextArea ta;
    String index, field, queries, queryString;
    int repeat;
```

```
boolean raw;
JScrollPane scroll;

public SearchFiles() {
    super("Search Engine");
    setLayout(null);
    t = new JTextField();
    t.setBounds(130, 40, 200, 40);
    add(t);
    b = new JButton("Submit");
    b.setBounds(180, 90, 100, 30);
    add(b);

    ta = new JTextArea();
    ta.setEditable(false);
    ta.setBounds(10, 140, 460, 600);
    ta.setSize(500,500);
    ta.setLineWrap(true);
    ta.setVisible(true);
    ta.setWrapStyleWord(true);
    add(ta);
    setSize(550, 800);
    setVisible(true);
    b.addActionListener(this);
}

public static void main(String[] args) {
    SearchFiles obj = new SearchFiles();
```

```
        String usage =
                "Usage:\tjava org.apache.lucene.demo.SearchFiles [-index dir] [-field f] [-repeat n] [-queries file] [-query string] [-raw] [-paging hitsPerPage]\n\nSee http://lucene.apache.org/core/4_1_0/demo/ for details.";
        if (args.length > 0 && ("-h".equals(args[0]) || "-help".equals(args[0]))) {
            obj.ta.append(usage+"\n");
            System.exit(0);
        }
    }
    void fun() throws Exception {
        String index = "G:\\Lucene\\Lucene-Tutorials-master\\test data";
        String field = "contents";
        String queries = null;
        int repeat = 0;
        boolean raw = false;
        String queryString = t.getText().trim();
        int hitsPerPage = 100;
        IndexReader reader = DirectoryReader.open(FSDirectory.open(Paths.get(index)));
        IndexSearcher searcher = new IndexSearcher(reader);
        Analyzer analyzer = new StandardAnalyzer();
        BufferedReader in = null;
        QueryParser parser = new QueryParser(field, analyzer);
        Query query = parser.parse(queryString);
        ta.append("Searching for: " + query.toString(field)+"\n");
        searcher.search(query, null, 100);
        doSearch(in, searcher, query, hitsPerPage, raw, queries == null && queryString == null);
        reader.close();
    }
```

```
public void doSearch(BufferedReader in, IndexSearcher searcher, Query query,
        int hitsPerPage, boolean raw, boolean interactive) throws IOException {

    TopDocs results = searcher.search(query, 5 * hitsPerPage);
    ScoreDoc[] hits = results.scoreDocs;

    int numTotalHits = results.totalHits;
    ta.setText(numTotalHits + " total matching documents\n");
    int start = 0;
    int end = Math.min(numTotalHits, hitsPerPage);

    for (int i = start; i < end; i++) {
        Document doc = searcher.doc(hits[i].doc);
        String path = doc.get("path");
        if (path != null) {
            ta.append((i + 1) + " " + path+"\n");
            String title = doc.get("title");
            if (title != null) {
                ta.append("   Title:" + doc.get("title")+"\n");
            }
        } else {
            ta.append((i + 1) + ". " + "No path for this document");
        }
    }
}

@Override
public void actionPerformed(ActionEvent e) {
```

```
        try {
            fun();
        } catch (Exception ex) {
            Logger.getLogger(SearchFiles.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

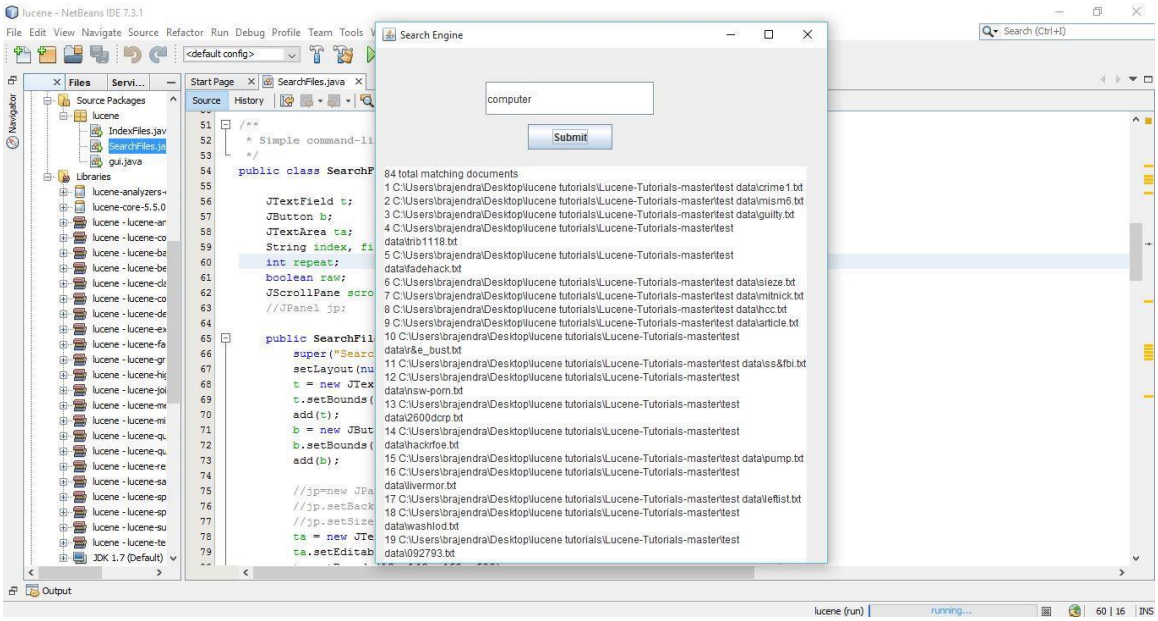

Figure 11: Input query and Result

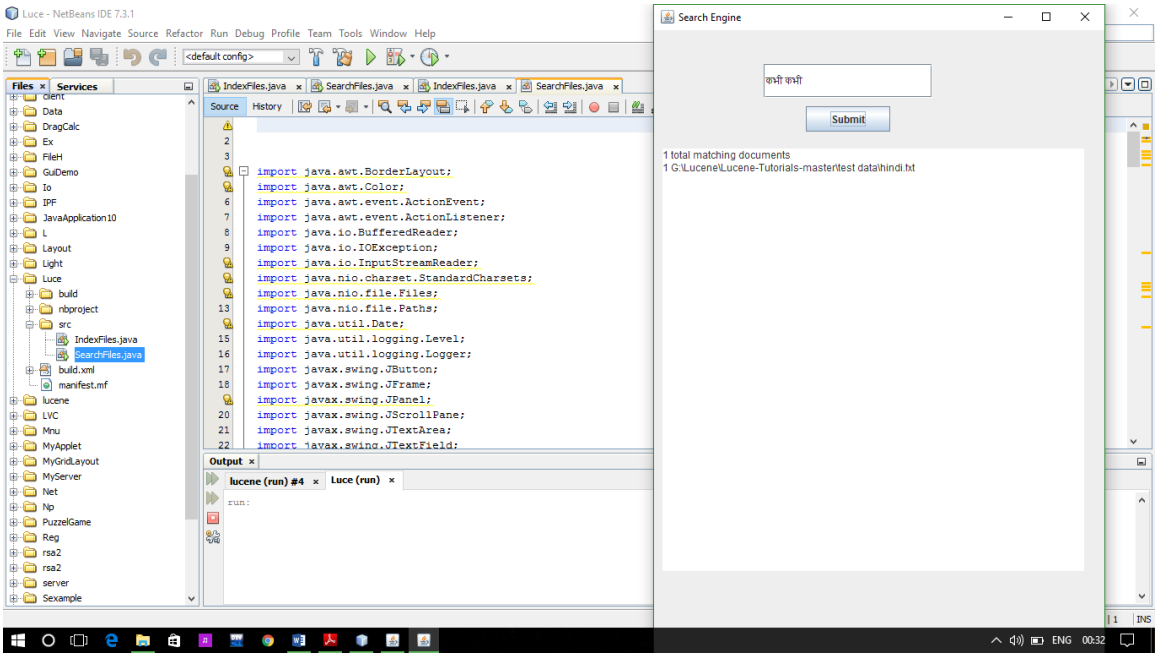

Figure 12: Multilingual search

Contribution of Each Member

Brajendra

- Proposed algorithm
- Implementation Details and Result
- Future Scope
- Setup Arrangement

Ratnesh Singh

- Literature Survey
- Comparative study
- Feature Encoding And Matching
- References

Gabbar Singh Sisodiya

- Study of Natural Language Processing
- Study of Information Retrieval
- System Testing And Performance Evaluation
- Conclusion