Web Caching

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Information Technology

under the Supervision of

Mr. Suman Saha

By

Raghav Verma (111461)

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "Web Caching", submitted by Raghav Verma, 111461 in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Mr. Suman Saha

Assistant Professor

Acknowledgement

I place on record and warmly acknowledge the continuous encouragement, invaluable supervision, timely suggestions and inspired guidance offered by my guide Mr. Suman Saha, Jaypee University of Information Technology (Waknaghat) in bringing this report to a successful completion.

Finally I extend my gratitude to one and all who are directly or indirectly involved in the successful completion of this project report.

Date: **Raghav Verma**

Table of Content

List of Figures

List of Tables

Abstract

The World Wide Web is a large distributed information system that provides access to shared data objects. The World Wide Web has an exponential growth rate, which results in network congestion and server overloading. And, this scorching rate of growth has put a heavy load on the Internet communication channels. This kind of situation is likely to continue in the future, as more and more information services move onto web. The result of this has escalated access latency for the users.

Access latency, which is the time interval between the user giving out a request and its actual time of completion, could result from many reasons. Servers gets flooded with more requests than they can handle optimally. The network path between the user and the server could become congested due to increased traffic on any or some of the constituent links.

Caching popular objects close to the users provides an opportunity to combat this latency by allowing users to fetch data from a nearby cache rather than from a distant server. Web caching is recognized as one of the effective schemes to alleviate the service bottleneck and reduce the network traffic, hence minimizes the user access latency.

The application works like adding all the file names into one file, so that record of all the files present in the directories can be maintained. Application converts the .pdf files to .txt files so as to index all the files. Inverted index is made corresponding to each word present in each file. Search operation can be performed on the indexed file so that it takes less time to search the particular word and the document in which the word is present.

Chapter 1: Introduction

1.1 Introduction

1.1.1 What is Web Caching?

Cache is memory that is stored very close to the CPU to allow fast access. A disk cache is memory that is used to store frequently accessed disk pages for fast access. Similarly, web caching is the storage of Web objects near the user so that they can have fast access, and it thus improves the user experience of the Web surfer.

So, we can define web cache as a temporary storage place for files requested from the Internet. After the request for data has been successfully made, and that data has been stored in the cache, further requests for those files results in the information being returned from the cache rather than the original location.

When a user visits a site such as www.xyz.com , Web caching will retrieve the page from the xyz.com Web server and store a copy of that page on the local server. The next time when a user requests www.xyz.com the Web cache will deliver the locally cached copy of the page. Through this, user will experience a very fast download because the request does not have to traverse the entire Internet – all the files will come from a local source or cache. And also, the bandwidth that would normally be used to download the Web site is not required and is free for other information retrieval or delivery.

Caching is useful for any library. For user's requests and saved bandwidth faster response is a good thing. Caching really makes sense for libraries that feel they must purchase more bandwidth to keep up with the increased usage. And in such cases a cache server or cache appliance could very likely lower the demand on the existing bandwidth, and thus making a costly bandwidth upgrade unnecessary.

1.1.2 Types of Web Caching

1) Browser cache:

It is located in the client. The user can notice the cache setting of any modern Web browser such as Internet Explorer Browser, Safari Browser, Mozilla Firefox Browser, Netscape Browser, and Google chrome Browser. Such cache is very useful, especially when users hit the ―back‖ button or click a link to see a page they have just looked at. In addition, if the user uses the same navigation images throughout the browsers, they will be most likely served from browsers' caches almost instantaneously.

2) Proxy server cache:

It is found in the proxy server which located between client machines and origin servers. It works on the same principle of browser cache, but only difference is that it works on a much larger scale. The browser cache which deals with only a single user, whereas the proxies serves hundreds or thousands of users in the same way. When a request is received, the proxy server checks in its cache. If it founds the object, it sends the object to the client. If the object is not found, or has expired, then the proxy server will request the object from the origin server and send it to the client. The object will be stored in the proxy's local cache for future requests.

3) Origin server cache:

Even at the origin server, web pages can be stored in a server-side cache for reducing the need for redundant computations or database retrievals. Therefore, the server load can only be reduced if the origin server cache is employed. Availability of Web access logs files that can be exploited as training data is the main motivation for adopting intelligent Web caching approaches. Another motivation, since the Web environment changes and updates rapidly and continuously, there is need of an efficient and adaptive scheme in Web environment. The machine learning techniques can adapt to the important changes through a training phase. Although there are many studies in Web caching, enhancement of Web caching performance using intelligent techniques is still fresh. Recent studies have shown that the intelligent approaches are more efficient and adaptive to the Web caching environment compared to other approaches.

1.1.3 Caching architectures

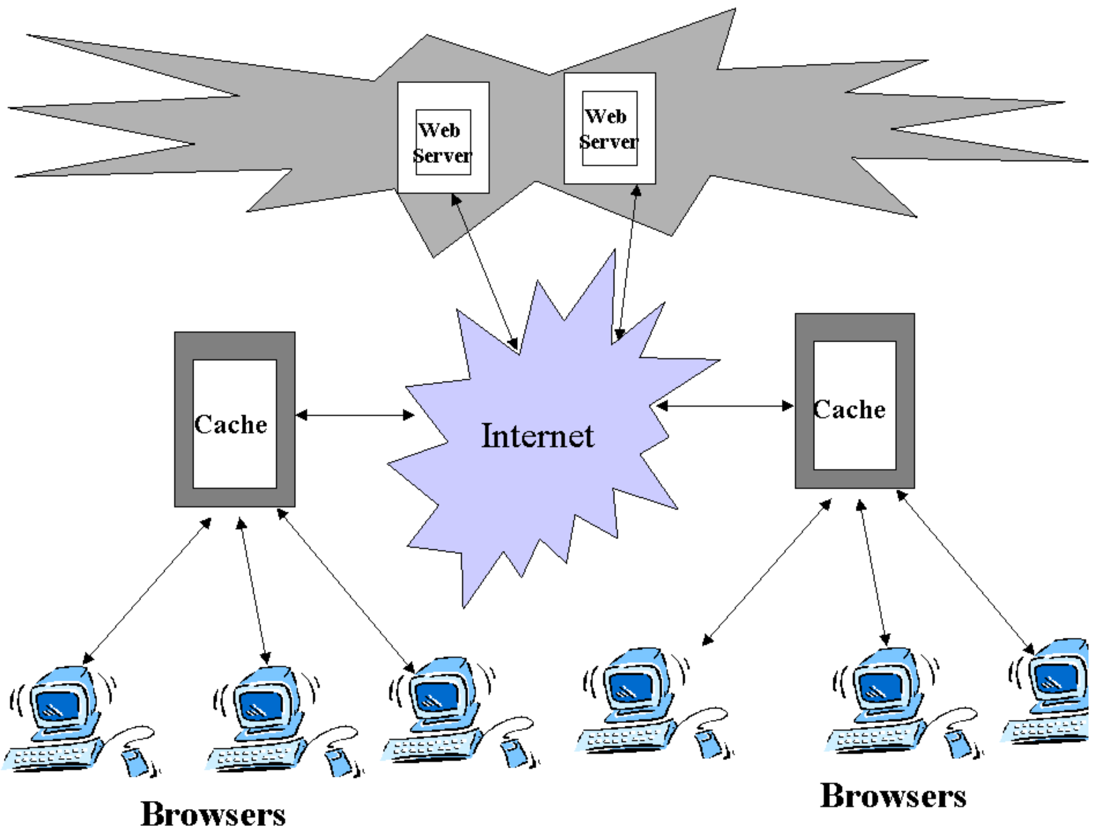


Fig 1: Caching Architecture

1.1.3.1 Hierarchical caching architecture

One approach to coordinate caches in the same system is to set up a caching hierarchy. With the help of hierarchical caching, caches can be placed at multiple levels of the network. For example, we assume that there are four levels of caches: bottom cache, institutional cache, regional cache, and national levels cache. At the bottom level cache of the hierarchy there are the client caches or browser caches. The request is redirected to the institutional cache when a request cannot be satisfied by the client cache. If the document is not available at the institutional level, the request is then most likely forwarded to the regional level cache which in turn forwards unsatisfied requests to the national level cache. If the document is not available at any cache level, then direct contact to the original server is done by national level cache. When the document is available, either at the original server or a cache, it will travel down the hierarchy, and will leave a copy at each of the intermediate caches which are along its path. More requests for the same document travel up the caching hierarchy until the document is hit at some cache level.

1.1.3.2 Distributed caching architecture

In distributed Web caching systems, there are no other intermediate cache levels than the institutional caches, as they serve each other's misses. To decide that from which institutional cache to retrieve a miss document, all institutional caches keep meta-data information about the content of every other institutional cache. With distributed caching, most of the traffic flows through low network levels, which are less congested. Moreover, distributed caching allows us better load sharing and are more fault tolerant. Nevertheless, a large-scale deployment of distributed caching may encounter several problems such as higher bandwidth usage, administrative issues, high connection time and few more.

There are several approaches to the distributed caching. Internet Cache Protocol (ICP), which supports discovery and retrieval of documents from neighbouring caches as well as parent caches. One other approach to the distributed caching is the Cache Array Routing protocol (CARP), which divides the URL-space among an array of loosely coupled caches and lets each cache store only the documents who's URL are hashed to it.

1.1.4 Cache placement/replacement

The key aspect of the effectiveness of proxy caches is a document placement/replacement algorithm that can yield high hit rate. In addition to traditional replacements policies like LRU, LFU there are other type of algorithms which try to minimize various cost metrics, such as the hit rate, the byte hit rate, the average latency, and the total cost. Current cache replacement strategies can be divided into 2 types.

1.1.4.1 Traditional replacement policies and its direct extensions:

- Least Recently Used (LRU): LRU evicts the object which was requested the least recently.
- Lease frequently used (LFU): LFU evicts the object which is accessed least frequently.

- Pitkow/Recker Strategy: This uses LRU. It evicts objects in the LRU order, except if all the objects are to be accessed within the same day, in which case the largest one is removed.

1.1.4.2 Key-based replacement policies:

- LRU-MIN: If there are any objects in the cache which have size being at least S, LRU-MIN evicts the least recently used such object from the cache. If no objects are there with size being at least S, then LRU-MIN starts evicting objects in LRU order of size being at least S/2. That is, the object who has the largest log (size) and is the least recently used object among all objects with the same log (size) will be evicted first.
- LRU-Threshold: It is the same as the LRU, but with the difference that objects that are larger than a certain threshold size are never cached.
- Lowest Latency First: It minimizes the average latency by evicting the document with the lowest download latency first.

1.1.3 Need of Web Caching

Caching helps us to bridge the performance gap between local activity and remote content. In other words, caching helps to improve Web performance by reducing the cost and end-user latency for Web access. In the long term, even as bandwidth costs continue to drop and higher end-user speeds become available, caching will help us continue to reap benefits for the following reasons:

- Bandwidth will always have some cost: The cost of bandwidth will never reach zero, even though increased competition, a growing market, and economies of scale reduce end-user costs. The cost of bandwidth at the core has stayed relatively stable, and requires ISPs to implement methods such as caching to stay competitive and reduce core bandwidth usage so that edge bandwidth costs can be low.
- Non-uniform bandwidth and latencies will persist: There will always be variations in bandwidth and latencies because of the physical limitations such as environment and location as well as the financial constraints. We can smooth these effects with the help of Caching.

- Increase in Network Distances: Firewalls, other proxies for security and privacy, and virtual private networks for telecommuters increase the number of hops through which content must travel and slow Web response times.
- Demand for Bandwidth continue to increase: Growth in the user base, in popularity of high-bandwidth media, and in user expectations of faster performance guarantee that demand for bandwidth will not end.
- Hot spots in the Web will continue: When high user demand for a site is predictable intelligent load balancing can alleviate problems, a Web site's popularity can also come as a result of current events, desirable content, or word of mouth. With help of Distributed Web caching we can alleviate these "hot spots" resulting from flash traffic loads.
- Communication costs exceed computational costs: Communication is likely to always be more expensive (to some extent) than computation. As CPUs are much faster than main memory therefore we use memory caches. Likewise, we will continue to use caches as computer systems and network connectivity both get faster.

1.1.4 Advantages of Web Caching

Documents can be cached on the clients, the proxies, and the servers. The effects of Web caching are two-fold. First, it has been shown that caching documents can improve Web performance significantly. There are several advantages of using Web caching:

- Web caching reduces bandwidth consumption, thereby decreases network traffic and lessens network congestion.
- Web caching reduces access latency due to two reasons:
  a) Frequently accessed documents are fetched from nearby proxy caches instead of remote data servers, the transmission delay is minimized.
  b) Because of the reduction in network traffic, those documents not cached can also be retrieved relatively faster than without caching due to less congestion along the path and less workload at the server.
- Web caching reduces the workload of the remote Web server by disseminating data among the proxy caches over the wide area network.

- If the remote server is not available due to the remote server's crash or network partitioning, the client can obtain a cached copy at the proxy. Thus, the robustness of the Web service is enhanced.
- A side effect of Web caching is that it provides us a chance to analyze an organization's usage patterns.

1.1.5 Disadvantages of Caching

- The main disadvantage is that a client might be looking at stale data due to the lack of proper proxy updating.
- The access latency may increase in the case of a cache miss due to the extra proxy processing. Hence, cache hit rate should be maximized and the cost of a cache miss should be minimized when designing a caching system.
- A single proxy cache is always a bottleneck. A limit has to be set for the number of clients a proxy can serve. An efficiency lower bound (i.e. the proxy system is ought to be at least as efficient as using direct contact with the remote servers) should also be enforced.
- Using a proxy cache will reduce the hits on the original remote server which might disappoint a lot of information providers, since they cannot maintain a true log of the hits to their pages. Hence, they might decide not to allow their documents to be cacheable.

1.2 Aim and Objectives

The project aims at developing an interactive application which can take input in form of text and can generate the output in the form of displaying the name of files corresponding to the text which is searched. It can be used for various purposes depending upon the situation to suit requirements of a programmer as well as a non-programmer human. It is developed in a manner so it can be used by anyone from a world famous IT Company to an ordinary man who barely knows how to operate the computer.

1.3 Project Vision

The vision of the project is to develop an application which comes hands on for all the people round the globe, it will help people to do their tasks with an ease and in a stress free manner. It will also provide support to differently abled people which face difficulty to have their hands – on the computer.

1.4 Motivation of the Project

The increasing growth of users over World Wide Web degrades the performance of system by network traffic and server heavy work load. In an organization, we see people connected to the same network for the use of internet. And everyone is unaware of others what they surf and download. So what happens is that the same content is downloaded many times which results into the consumption of network bandwidth.

Chapter 2: Literature Survey

2.1 Summary of papers

Title	Survey on Integrating Web Caching and Pre-Fetching
Author	T.V. Sree Ramya and V. Sathiyamoorthi
Year	4, April 2013
Source	International Journal of Advanced Research in Computer Science and Software Engineering
Summary	Web caching and prefetching are two effective solutions to lessen Web service bottleneck, moreover reduce traffic over the Internet and also improve scalability of the Web system. The Web prefetching and Web caching complements each other since the web caching exploits the temporal locality for predicting revisiting requested objects, whereas the web prefetching utilizes the spatial locality for predicting next related web objects of the requested Web objects. Thus, combination of the web caching and the web prefetching doubles the performance compared to single caching. This paper reviews some principles and some of the existing web caching and prefetching approaches. Firstly, we have reviewed principles and existing works of web caching. The conventional web caching and intelligent web caching are included in this. And secondly, types of prefetching and categories of prefetching have presented and discussed briefly. Moreover, the history-based prefetching approaches have been concentrated and discussed with review of the related works for each approach in this survey.

Table 1: Survey on Integrating Web Caching and Pre-Fetching paper summary

Title	Performance Analysis of Web Caching Through Cache Replacement Based on User Behavior
Author	Anita Madaan, Niraj Madaan and Poonam Jain
Year	12, December 2012
Source	International Journal of Advanced Research in Computer Science and Software Engineering
Summary	Web traffic has more structure to it than most type of internet traffic. As browsers and servers evolve the characteristics of the traffic change, as the behaviour of user changes, and the protocols change it increase the speed of the network. Frequently-retrieved documents are stored through caching proxy into the local files for future use, so it will not be necessary to download the document next time whenever it is requested in future. Caching proxies save network traffic and reduce Web latency by storing popular documents closer to the users, High variability is an invariant in caching response times across log data of different proxies. Through a combination of factors such as the high variability in file sizes and bandwidth of the client links to the caching proxies' high variability can be explained. By using web caching traffic load decreases 50% and frequency distribution is made faster. A significant section of cacheable Web traffic is static, this means that it can be predicted for a long period of time based on the past observations. However, about 25% of the Web traffic consists of HTML and other "ASCII" documents. These documents have an average compression ratio of 60%. An algorithm may consult a pre-computed table of average compression ratios to decide if the compression of a particular document type is beneficial.

Table 2: Performance Analysis of Web Caching Through cache replacement based on user behaviour paper summary

Title	A Web Caching Primer
Author	Davison, B.D.
Year	2001
Source	Internet Computing, IEEE
Summary	Caching helps bridge the performance gap between local activity and remote content. In other words, caching helps us to improve Web performance by reducing the cost and end-user latency for the Web access. Also in the long term, even as the bandwidth costs continue to drop and higher end-user speeds become available, caching will help us continue to reap benefits. Given the choices of caching products on the market today, how do you select one? The process involves determining the features and performance level (such as requests per second, bandwidth saved, and average latency, often measured by cache benchmarking services) we require. Moreover, as caches and content delivery services replace origin servers in serving Web traffic, content developers should consider how to maximize the usefulness of these technologies.

Table 3: A Web Caching Primer paper summary

Title	Site-Based Approach to Web Cache Design
Author	Kin Yeung Wong, Kai Hau Yeung
Year	2001
Source	IEEE
Summary	There are three popular research areas in Web caching: cache replacement policy, prefetching, and multiple cache server design. Another less active but important area of research is proxy load reduction. Proxy load reduction is crucial because as Web traffic increases, so too do the number of client requests reaching the proxy server. Cache replacement *policy* evaluates existing cached documents to determine which documents should be replaced when the cache is full and new documents arrive. Prefetching individually examines each hyperlink on a Web page to reduce the latency user's experience. Multiple cache server design (multi-cache design) focuses on approaches for exchanging caching information and redirecting requests or documents between cache servers. Our proposed site-based approach brings many benefits to different areas of the Web caching. One problem, is that while we maintain only site information it is sufficient in many of the cases, it is sometimes necessary to also track individual documents. For example, although the site-based LRU purges a whole site when a replacement of cache is required, the system might still be needing information on individual cached documents. Currently, there is an attemp to modify a real proxy server to use the site-based approach.

Table 4: Site-Based Approach to Web Cache Design paper summary

Chapter 3: Application Design

3.1 Introduction

This application aims at developing a highly – efficient software which would cater to the needs of the people of the programming sector. It can be used for various purposes depending upon the situation to suit requirements of a programmer as well as a non-programmer human. It is developed in a manner so it can be used by anyone from a world famous IT Company to an ordinary man who barely knows how to operate the computer.

3.2 Use Case Modelling

It helps us to describe the functional requirements of the application developed in the overall system.

Use Case: File Search

Primary Actor: User

Goal: To search the file corresponding to the text.

Use Case Description:

Actor: User

Pre-Condition: The user must know how to operate and type on computer.

If the user clicks the application then the application should open up. When the users enters some text to search then the text is passed to the index file. If the text is present in the index file then the corresponding file name is returned else the text is searched in the internet and the respective file is being downloaded and added to the index file and then the result is displayed to the user.

Post-Condition: The user must know how to save a file and must be able to access the file in future for further use or modification.

Use Case Diagram:

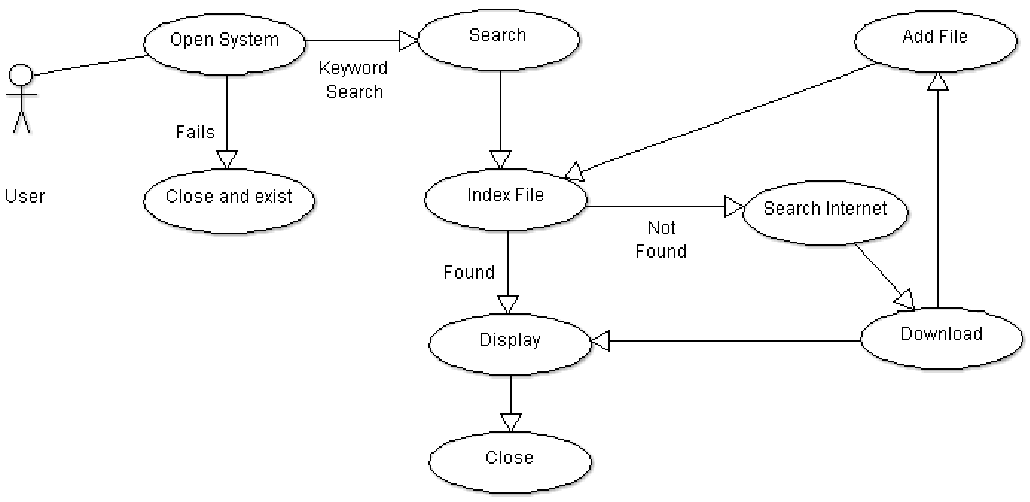


Fig 2: Use Case Diagram

3.3 Class Oriented Model

This helps developer to represent a system in form of object oriented manner in which classes collaborate to achieve the system requirements.

Classes: Search

Class Diagram:

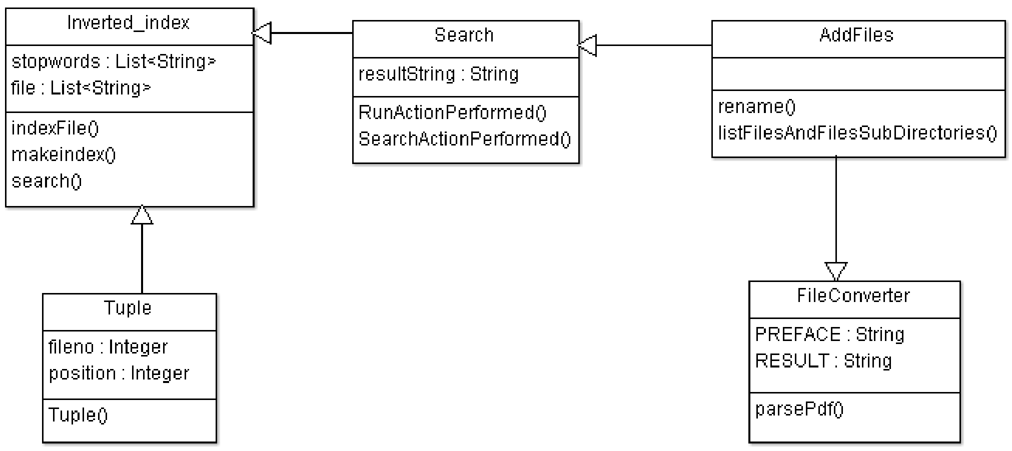


Fig 3: Class Diagram

3.4 Requirement Analysis Models

Through this phase we find the requirements more in detailed manner by examining their detailed boundary conditions & exceptional cases.

3.4.1 Data Flow Diagram:

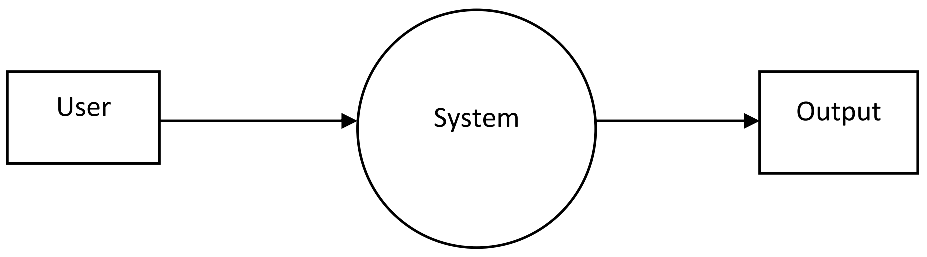
These diagrams depict the flow of data from one point of the system to another point. It mainly consists of three parts divided on the basis of its level:

a) *Level '0' Data Flow Diagram (Context Diagram).*

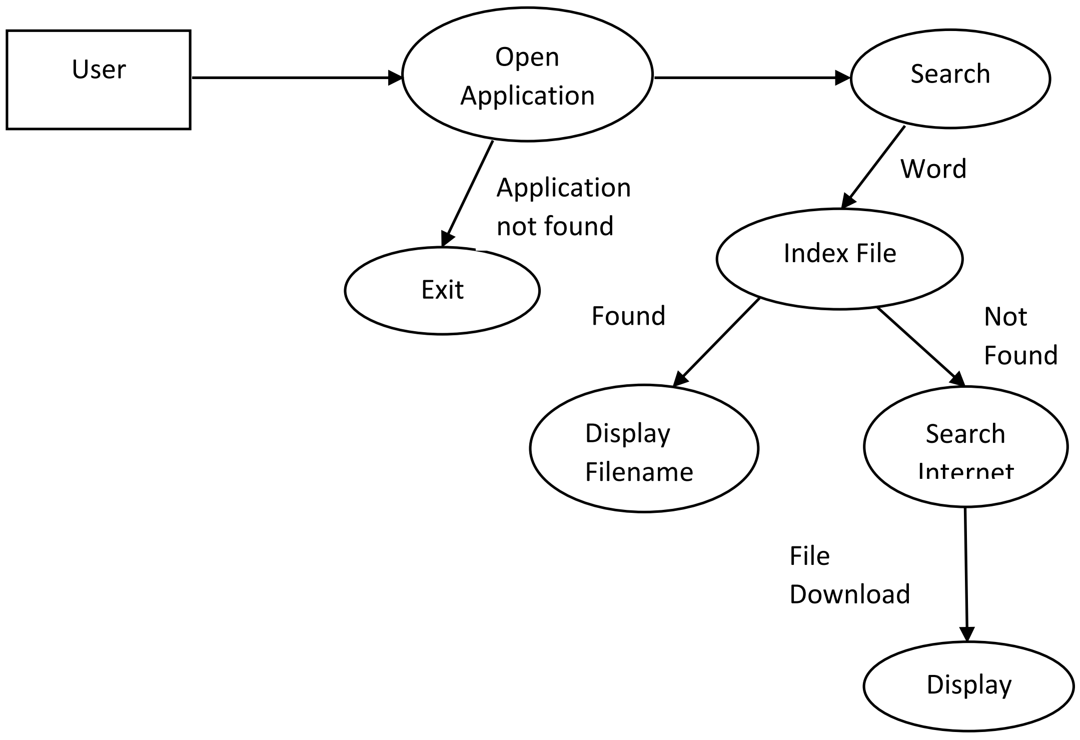
b) *Level '1' DFD.*

c) *Level '2' DFD.*

Level '0' DFD or Context Diagram:



Level '1' DFD:



Level '2' DFD:

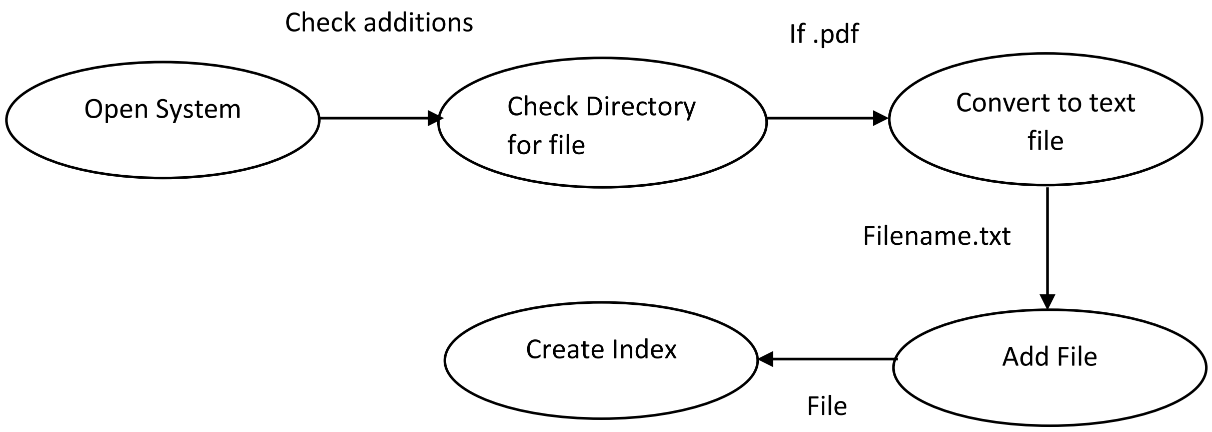
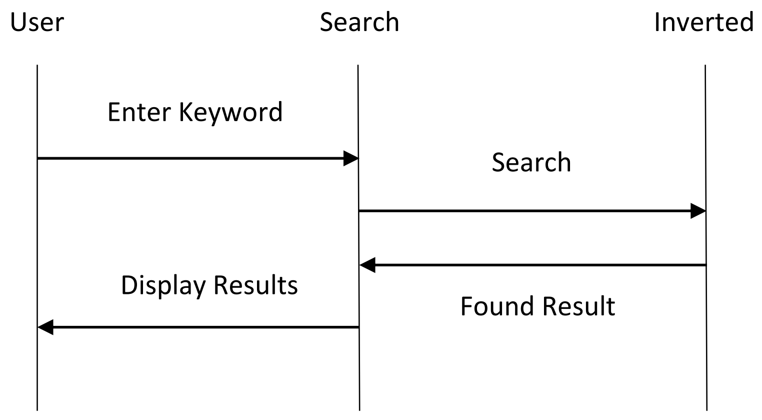


Fig 4: Data Flow Diagram

3.4.2 Behaviour Modelling:

It tells us about how system behaves & how users use it. In this we have to use 'Use-Cases' & 'Use-Case Description' for which you have to develop steps. It includes two types of Diagrams.

Sequence Diagrams:



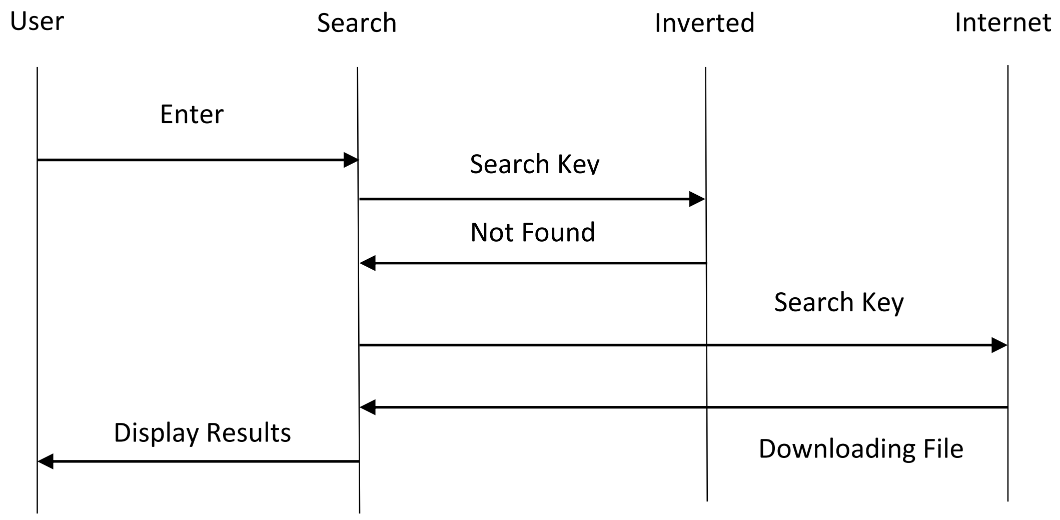


Fig 5: Sequence Diagram

3.4.3 State Diagram:

It is useful when an object has a dynamic behaviour. It specifies the sequence of states in which an object can exist along with the events and conditions that causes transition between the states.

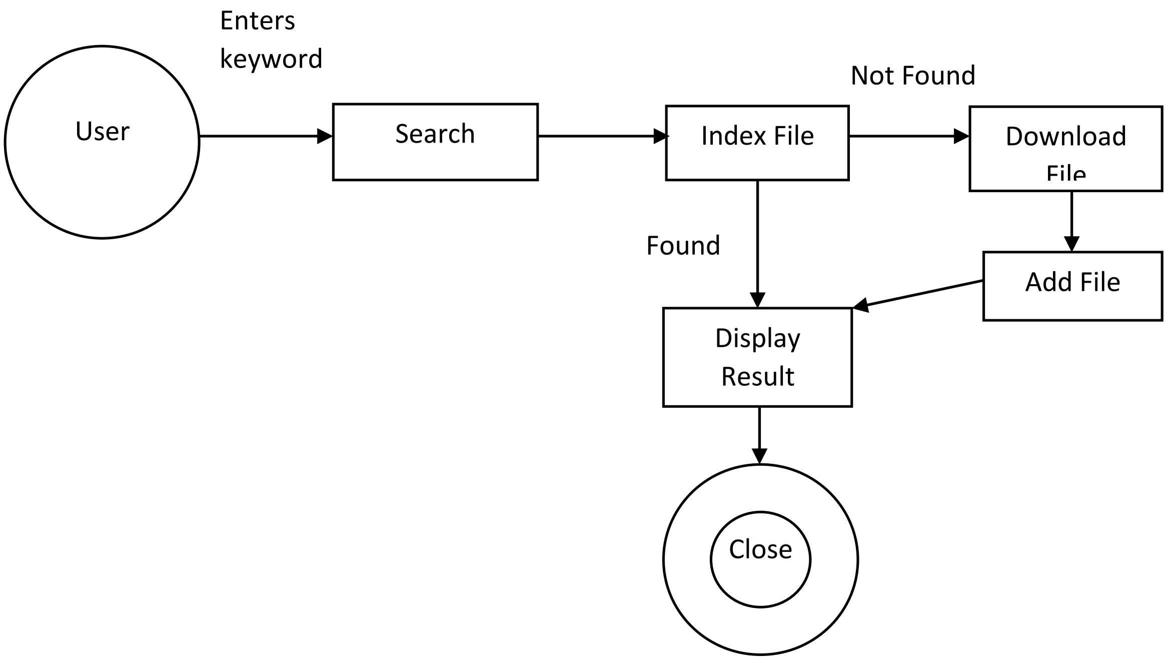


Fig 6: State Diagram

3.5 Usefulness and Innovation

This software helps us to maintain the caching system on our machine. Client sends the request, the request is passed to the index file. It searches the data in the index file. If the data is found then the result is returned back to the user. Anyhow if the data is not found in the file then the file is automatically downloaded from the internet and the result is returned to user. In order to maximize the hit rate we have to maximize the number of requests that have been satisfied in the cache. It depends on cache size as the cache size increases more hit rates get. When the hit rates maximize users get the results with minimum delay. The caching system finally upgrades the performance. It reduces the traffic and improves the access latency.

Chapter 4: Search Engine

Search engine indexing collects, parses, and stores data to facilitate fast and accurate information retrieval. Index design incorporates interdisciplinary concepts from linguistics, cognitive psychology, mathematics, informatics, and computer science. An alternate name for the process in the context of search engines designed to find web pages on the Internet is web indexing.

Popular engines focus on the full-text indexing of online, natural language documents. Media types such as video and audio and graphics are also searchable.

Meta search engines reuse the indices of other services and do not store a local index, whereas cache-based search engines permanently store the index along with the corpus. Unlike full-text indices, partial-text services restrict the depth indexed to reduce index size. Larger services typically perform indexing at a predetermined time interval due to the required time and processing costs, while agent-based search engines index in real time.

4.1 Indexing

The purpose of storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power. For example, while an index of 10,000 documents can be queried within milliseconds, a sequential scan of every word in 10,000 large documents could take hours. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval.

4.1.1 Index design factors

Major factors in designing a search engine's architecture include:

Merge factors

How data enters the index, or how words or subject features are added to the index during text corpus traversal, and whether multiple indexers can work asynchronously. The indexer must first check whether it is updating old content or adding new content.

Traversal typically correlates to the data collection policy. Search engine index merging is similar in concept to the SQL Merge command and other merge algorithms.

Storage techniques

How to store the index data, that is, whether information should be data compressed or filtered.

Index size

How much computer storage is required to support the index?

Lookup speed

How quickly a word can be found in the inverted index. The speed of finding an entry in a data structure, compared with how quickly it can be updated or removed, is a central focus of computer science.

Maintenance

How the index is maintained over time.

Fault tolerance

How important it is for the service to be reliable. Issues include dealing with index corruption, determining whether bad data can be treated in isolation, dealing with bad hardware, partitioning, and schemes such as hash-based or composite partitioning, as well as replication.

4.1.2 Index data structures

Search engine architectures vary in the way indexing is performed and in methods of index storage to meet the various design factors.

Suffix tree

Figuratively structured like a tree, supports linear time lookup. Built by storing the suffixes of words. The suffix tree is a type of trie. Tries support extendable hashing, which is important for search engine indexing. Used for searching for patterns in DNA sequences and clustering. A major drawback is that storing a word in the tree may require space beyond that required to store the word itself. An alternate representation is a suffix array, which is considered to require less virtual memory and supports data compression such as the BWT algorithm.

Inverted index

Stores a list of occurrences of each atomic search criterion, typically in the form of a hash table or binary tree.

Citation index

Stores citations or hyperlinks between documents to support citation analysis, a subject of Bibliometrics.

Ngram index

Stores sequences of length of data to support other types of retrieval or text mining.

Document-term matrix

Used in latent semantic analysis, stores the occurrences of words in documents in a two-dimensional sparse matrix.

4.1.3 Compression

Generating or maintaining a large-scale search engine index represents a significant storage and processing challenge. Many search engines utilize a form of compression to reduce the size of the indices on disk. Consider the following scenario for a full text, Internet search engine.

- It takes 8 bits (or 1 byte) to store a single character. Some encodings use 2 bytes per character.
- The average number of characters in any given word on a page may be estimated at 5

Given this scenario, an uncompressed index (assuming a non-conflated, simple, index) for 2 billion web pages would need to store 500 billion word entries. At 1 byte per character, or 5 bytes per word, this would require 2500 gigabytes of storage space alone. This space requirement may be even larger for a fault-tolerant distributed storage architecture. Depending on the compression technique chosen, the index can be reduced to a fraction of this size. The trade-off is the time and processing power required to perform compression and decompression.

Notably, large scale search engine designs incorporate the cost of storage as well as the costs of electricity to power the storage. Thus compression is a measure of cost.

4.2 Working of Search Engine

A search engine operates in the following order:

1. Web crawling
2. Indexing
3. Searching

Web search engines work by storing information about many web pages, which they retrieve from the HTML markup of the pages. These pages are retrieved by a Web crawler (sometimes also known as a spider) — an automated Web crawler which follows every link on the site.

The search engine then analyses the contents of each page to determine how it should be indexed (for example, words can be extracted from the titles, page content, headings, or special fields called meta tags). Data about web pages are stored in an index database for use in later queries. A query from a user can be a single word. The index helps find information relating to the query as quickly as possible. Some search engines, such as Google, store all or part of the source page (referred to as a cache) as well as information about the web pages, whereas others, such as AltaVista, store every word of every page they find. This cached page always holds the actual search text since it is the one that was actually indexed, so it can be very useful when the content of the current page has been updated and the search terms are no longer in it. This problem might be considered a mild form of linkrot, and Google's handling of it increases usability by satisfying user expectations that the search terms will be on the returned webpage. This satisfies the principle of least astonishment, since the user normally expects that the search terms will be on the returned pages. Increased search relevance makes these cached pages very useful as they may contain data that may no longer be available elsewhere.

When a user enters a query into a search engine (typically by using keywords), the engine examines its index and provides a listing of best-matching web pages according to its criteria, usually with a short summary containing the document's title and sometimes parts of the text. The index is built from the information stored with the data and the method by which the information is indexed. From 2007 the Google.com search engine has allowed one to search by date by clicking "Show search tools" in the leftmost column of the initial search results page, and then selecting the desired date range. Most search engines support the use of the Boolean operators AND, OR and NOT to further specify the search query. Boolean operators are for literal searches that allow the user to refine and extend the terms of the search. The engine looks for the words or phrases exactly as entered. Some search engines provide an advanced feature called proximity search, which allows users to define the distance between keywords. There is also concept-based searching where the research involves using statistical analysis on pages containing the words or phrases you search for. As well, natural language queries allow the user to type a question in the same form one would ask it to a human. A site like this would be ask.com.

The usefulness of a search engine depends on the relevance of the result set it gives back. While there may be millions of web pages that include a particular word or phrase, some pages may be more relevant, popular, or authoritative than others. Most search engines employ methods to rank the results to provide the "best" results first. How a search engine decides which pages are the best matches, and what order the results should be shown in, varies widely from one engine to another. The methods also change over time as Internet usage changes and new techniques evolve. There are two main types of search engine that have evolved: one is a system of predefined and hierarchically ordered keywords that humans have programmed extensively. The other is a system that generates an "inverted index" by analysing texts it locates. This first form relies much more heavily on the computer itself to do the bulk of the work.

Most Web search engines are commercial ventures supported by advertising revenue and thus some of them allow advertisers to have their listings ranked higher in search results for a fee. Search engines that do not accept money for their search results make money by running search related ads alongside the regular search engine results. The search engines make money every time someone clicks on one of these ads.

Chapter 5: Inverted Index

5.1 Introduction

Using an index we can increase the speed and efficiency of searches of the document collection. Without some sort of index, a user's query must sequentially scan the document collection, finding those documents containing the search terms. Consider the "Find" operation in Windows; a user search is initiated and a search starts through each file on the hard disk. When a directory is encountered, the search continues through each directory. With only a few thousand files on a typical laptop, a typical "find" operation takes a minute or longer. Assume we have a fast processor executing an efficient sequential search algorithm. Further, assume we have a fast processor and high speed disk drives that enable a scan of one million documents per second (figure a document to be 4KB). This implies a scan rate of 400 GB per second, a generous processing assumption. Currently, a web search covers at least one billion documents. This back of the envelope calculation puts us at around 1000 seconds or a tad longer than sixteen minutes (and we were generous in estimating the speed of the scan). Users are not going to wait sixteen minutes for a response. Hence, a sequential scan is simply not feasible.

B-trees (balanced trees) were developed for efficient database access in the 1970's [B-tree ref]. A survey of B-tree algorithms is found in [xxx]. The B-tree is designed to remain balanced even after numerous updates. This means that even in the presence of updates, retrieval remains at a speed of O (logn) where n is the number of keys. Hence, update is well balanced with retrieval, both require O (logn).

Within the search engine domain, data are searched far more frequently than they are updated. Certainly, users make changes to documents, but it is not clear that a searcher requires the latest change to satisfy most search requests. Given this situation a data structure called an inverted index is commonly used by search engines.

An inverted index is able to do many accesses in O (1) time at a price of significantly longer time to do an update, in the worst case O (n). Index construction time is longer as well, but query time is generally faster than with a b-tree. Since index construction is an off- line process, shorter query processing times at the expense of lengthier index construction times is an appropriate trade off.

Finally, inverted index storage structures can exceed the storage demands of the document collection itself. However, for many systems, the inverted index can be compressed to around ten percent of the original document collection. Given the alternative (of twenty minute searches), search engine developers are happy to trade index construction time and storage for query efficiency.

An inverted index is an optimized structure that is built primarily for retrieval, with update being only a secondary consideration. The basic structure inverts the text so that instead of the view obtained from scanning documents where a document is found and then its terms are seen (think of a list of documents each pointing to a list of terms it contains), an index is built that maps terms to documents (pretty much like the index found in the back of this book that maps terms to page numbers). Instead of listing each document once (and each term repeated for each document that contains the term), an inverted index lists each term in the collection only once and then shows a list of all the documents that contain the given term. Each document identifier is repeated for each term that is found in the document.

An inverted index contains two parts: an index of terms, (generally called simply the term index, lexicon, or term dictionary) which stores a distinct list of terms found in the collection and, for each term, a posting list, a list of documents that contain the term.
Consider the following two documents:
D1: The GDP increased 2 percent this quarter.
D2: The spring economic slowdown continued to spring downwards this quarter.
An inverted index for these two documents is given below:

2 -> [D1]
continued -> [D2]
downwards -> [D2]
economic -> [D2]
GDP -> [D1]

increased -> [D1]

percent -> [D1]

quarter -> [D1] -> [D2]

slowdown -> [D2]

spring -> [D2]

the -> [D1] -> [D2]

this -> [D1] -> [D2]

to -> [D2]

As shown, the terms continued, economic, slowdown, spring, and to appear in only the second document, the terms GDP, increased, and percent, and the numeral 2 appear in only the first document, and the terms quarter, the, and this appear in both documents.

5.2 Applications of Inverted Index

The inverted index data structure is a central component of a typical search engine indexing algorithm. A goal of a search engine implementation is to optimize the speed of the query: find the documents where word X occurs. Once a forward index is developed, which stores lists of words per document, it is next inverted to develop an inverted index. Querying the forward index would require sequential iteration through each document and to each word to verify a matching document. The time, memory, and processing resources to perform such a query are not always technically realistic. Instead of listing the words per document in the forward index, the inverted index data structure is developed which lists the documents per word.

With the inverted index created, the query can now be resolved by jumping to the word id (via random access) in the inverted index.

In pre-computer times, concordances to important books were manually assembled. These were effectively inverted indexes with a small amount of accompanying commentary that required a tremendous amount of effort to produce.

In bioinformatics, inverted indexes are very important in the sequence assembly of short fragments of sequenced DNA. One way to find the source of a fragment is to search for it against a reference DNA sequence. A small number of mismatches (due to differences between the sequenced DNA and reference DNA, or errors) can be accounted for by dividing the fragment into smaller fragments—at least one sub fragment is likely to match the reference DNA sequence. The matching requires constructing an inverted index of all substrings of a certain length from the reference DNA sequence. Since the human DNA contains more than 3 billion base pairs, and we need to store a DNA substring for every index, and a 32-bit integer for index itself, the storage requirement for such an inverted index would probably be in the tens of gigabytes.

5.3 Application Interface

Application name is Caching Software. As we can see, the interface has five buttons that is download history button, download links button, add files button, and create index file button and the last search query button. Interface also has a text field where we can write our query so as to search for the files that contain the text. Text area is also a part of interface where the result will be displayed when we will download history, download links, add files and create inverted index file. JList is also used where the search query output will be displayed.

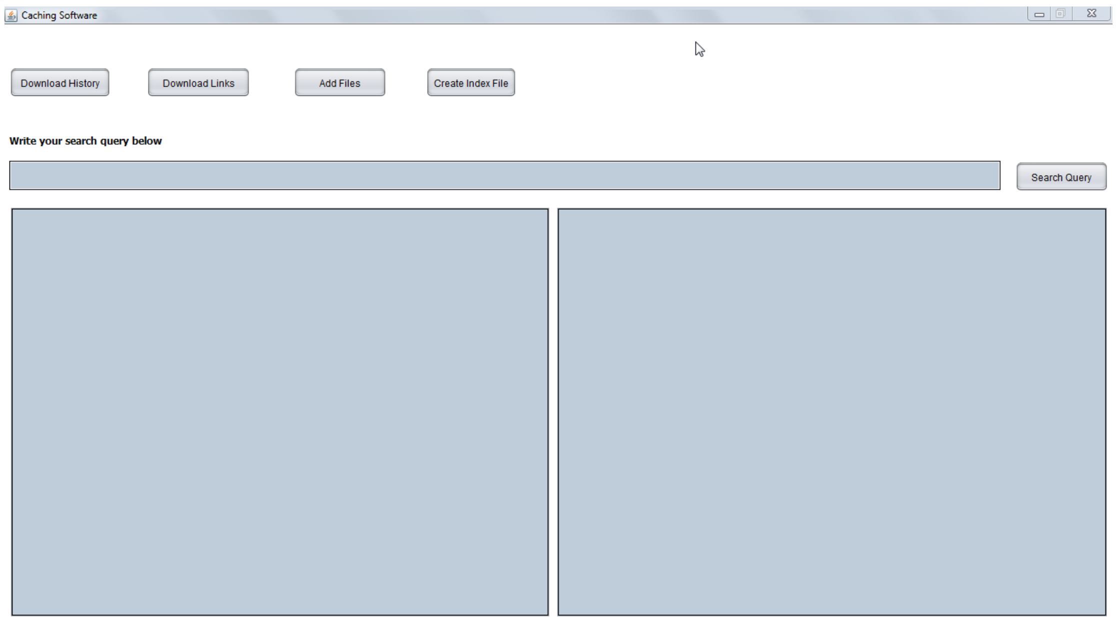
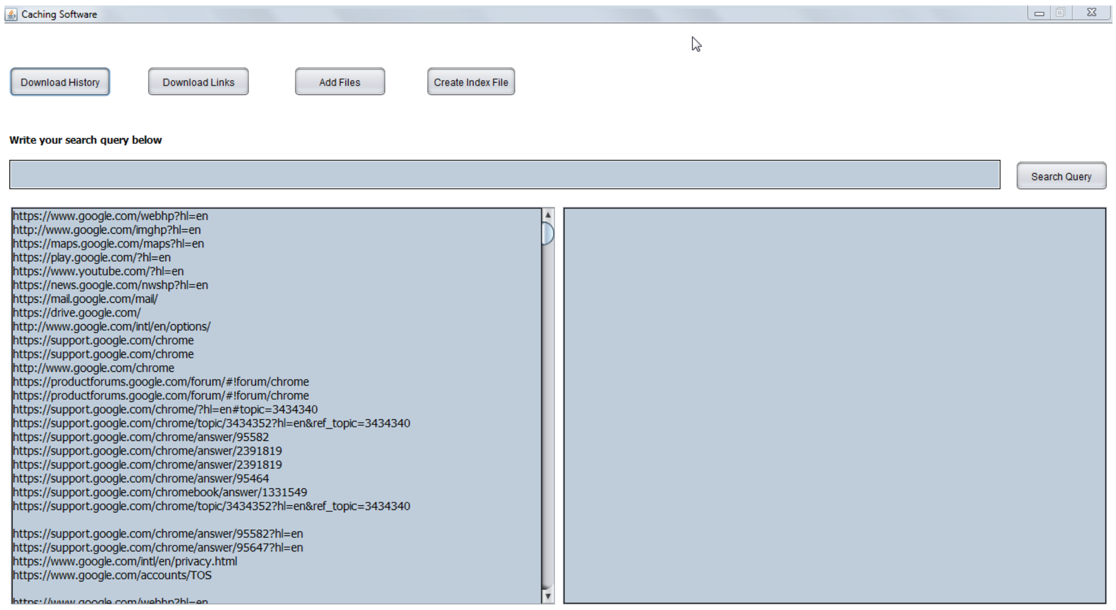


Fig 7: Application Interface

5.3.1 Download History

Download history is a button when clicked will create an object for the Download_History class and then with the help of this object it will call the downloadlinks method. In this method a history file is opened up and all the links present in this html file are copied to the history.txt file. Now history.txt file is read line by line and the links are passed to the Extract method present in the Extract_Links class. In this method it takes the link connect to it using Jsoup and selects all the links present in this web page. Now the links are checked in the hashmap so as to check if they exist then the link won't be added else the link will be added to the hashmap. After doing all this we serialize the file. We simply write the hashmap to the serialized file that is hashmap.ser.



```
https://www.google.com/webhp?hl=en
http://www.google.com/imghp?hl=en
https://maps.google.com/maps?hl=en
https://play.google.com/?hl=en
https://www.youtube.com/?hl=en
https://news.google.com/nwshp?hl=en
https://mail.google.com/mail/
https://drive.google.com/
http://www.google.com/intl/en/options/
https://support.google.com/chrome
https://support.google.com/chrome
http://www.google.com/chrome
https://productforums.google.com/forum/#!forum/chrome
https://productforums.google.com/forum/#!forum/chrome
https://support.google.com/chrome/?hl=en#topic=3434340
https://support.google.com/chrome/topic/3434352?hl=en&ref_topic=3434340
https://support.google.com/chrome/answer/95582
https://support.google.com/chrome/answer/2391819
https://support.google.com/chrome/answer/2391819
https://support.google.com/chrome/answer/95464
https://support.google.com/chromebook/answer/1331549
https://support.google.com/chrome/topic/3434352?hl=en&ref_topic=3434340

https://support.google.com/chrome/answer/95582?hl=en
https://support.google.com/chrome/answer/95647?hl=en
https://www.google.com/intl/en/privacy.html
https://www.google.com/accounts/TOS

https://www.google.com/webhp?hl=en
```

Fig 8: Download History Output

So in the above figure we see the output that is the links which are fetched from each web page and is displayed in the textarea. This is what download_links function looks like in the Download_History class :

```
void downloadlinks() throws IOException
{
        File input = new File("saved_resource.html");
```

```
         Document doc = Jsoup.parse(input, "UTF-8");
         Elements links = doc.select("a[href]");
}
```

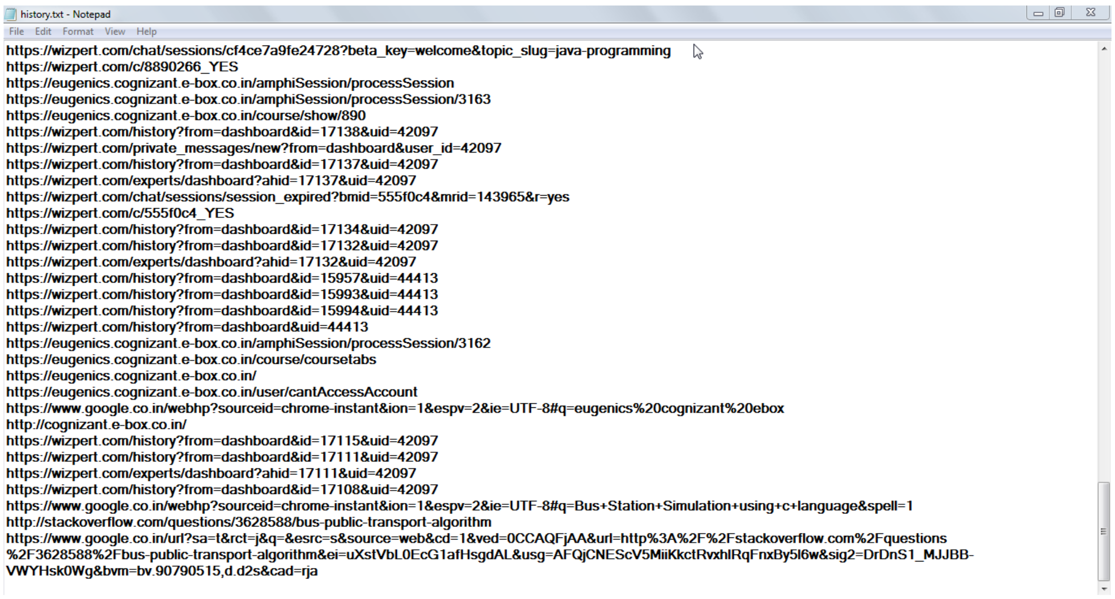
Now we use the for loop to write the links to the history.txt file.This the code snippet to extract the links from the given url.

```
public static void Extract(String text) throws IOException
{
        if(links_store.containsKey(text))
            return;
        else if(links_store.containsKey(null))
            return;
        else
        {
            links_store.put(text,i);
            i++;
            String url = text;
            Document doc =
Jsoup.connect(url).timeout(10*1000000000).ignoreContentType(true).get();

             Elements links = doc.select("a[href]");
             for (Element link : links)
             {
                 if(links_store.containsKey(link))
                                     continue;
                 else
                 {
                 print(link.attr("abs:href"), trim(link.text(), 35));
                  }
             }
            }
            }
```

This is the code snippet for serializing the hashmap.

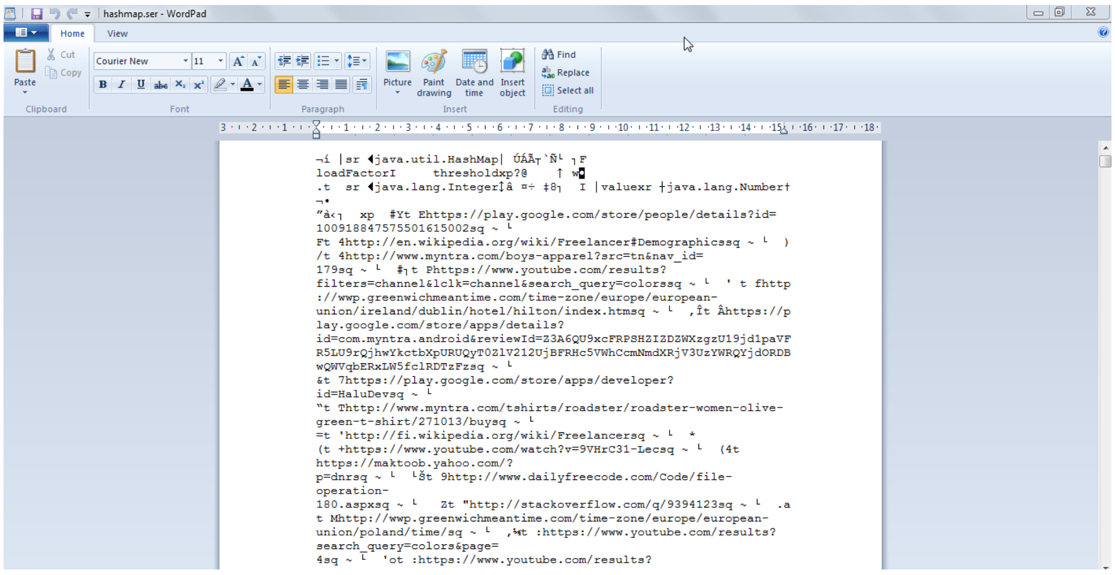
```
public static void Write_File()
{
HashMap<String,Integer> wf=new HashMap<String,Integer>();
wf.putAll(Extract_Links.links_store);
try
{
FileOutputStream fos =new FileOutputStream("hashmap.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(wf);
 oos.close();
fos.close();
}
```



```
https://wizpert.com/chat/sessions/cf4ce7a9fe24728?beta_key=welcome&topic_slug=java-programming
https://wizpert.com/c/8890266_YES
https://eugenics.cognizant.e-box.co.in/amphiSession/processSession
https://eugenics.cognizant.e-box.co.in/amphiSession/processSession/3163
https://eugenics.cognizant.e-box.co.in/course/show/890
https://wizpert.com/history?from=dashboard&id=17138&uid=42097
https://wizpert.com/private_messages/new?from=dashboard&user_id=42097
https://wizpert.com/history?from=dashboard&id=17137&uid=42097
https://wizpert.com/experts/dashboard?ahid=17137&uid=42097
https://wizpert.com/chat/sessions/session_expired?bmid=555f0c4&mrid=143965&r=yes
https://wizpert.com/c/555f0c4_YES
https://wizpert.com/history?from=dashboard&id=17134&uid=42097
https://wizpert.com/history?from=dashboard&id=17132&uid=42097
https://wizpert.com/experts/dashboard?ahid=17132&uid=42097
https://wizpert.com/history?from=dashboard&id=15957&uid=44413
https://wizpert.com/history?from=dashboard&id=15993&uid=44413
https://wizpert.com/history?from=dashboard&id=15994&uid=44413
https://wizpert.com/history?from=dashboard&uid=44413
https://eugenics.cognizant.e-box.co.in/amphiSession/processSession/3162
https://eugenics.cognizant.e-box.co.in/course/coursetabs
https://eugenics.cognizant.e-box.co.in/
https://eugenics.cognizant.e-box.co.in/user/cantAccessAccount
https://www.google.co.in/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=eugenics%20cognizant%20ebox
http://cognizant.e-box.co.in/
https://wizpert.com/history?from=dashboard&id=17115&uid=42097
https://wizpert.com/history?from=dashboard&id=17111&uid=42097
https://wizpert.com/experts/dashboard?ahid=17111&uid=42097
https://wizpert.com/history?from=dashboard&id=17108&uid=42097
https://www.google.co.in/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=Bus+Station+Simulation+using+c+language&spell=1
http://stackoverflow.com/questions/3628588/bus-public-transport-algorithm
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCAQFjAA&url=http%3A%2F%2Fstackoverflow.com%2Fquestions
%2F3628588%2Fbus-public-transport-algorithm&ei=uXstVbL0EcG1afHsgdAL&usg=AFQjCNEScV5MiiKkctRvxhlRqFnxBy5l6w&sig2=DrDnS1_MJJBB-
VWYHsk0Wg&bvm=bv.90790515,d.d2s&cad=rja
```

Fig 9: History.txt file

In the above figure we see the links that are fetched from the browser history and are stored in the history.txt file.



```
¬í |sr ◀java.util.HashMap| ÚÁÃ┬`Ñ└ ┐F
loadFactorI     thresholdxp?@    ↑ w
.t  sr ◀java.lang.Integer↕â ¤÷ ‡8┐  I |valuexr ┼java.lang.Number†
¬•
”à‹┐  xp  #Yt Ehttps://play.google.com/store/people/details?id=
100918847575501615002sq ~ └
Ft 4http://en.wikipedia.org/wiki/Freelancer#Demographicssq ~ └  )
/t 4http://www.myntra.com/boys-apparel?src=tn&nav_id=
179sq ~ └  #┐t Phttps://www.youtube.com/results?
filters=channel&lclk=channel&search_query=colorssq ~ └  ' t fhttp
://wwp.greenwichmeantime.com/time-zone/europe/european-
union/ireland/dublin/hotel/hilton/index.htmsq ~ └  ,Ît Âhttps://p
lay.google.com/store/apps/details?
id=com.myntra.android&reviewId=Z3A6QU9xcFRPSHZIZDZWXzgzU19jd1paVF
R5LU9rQjhwYkctbXpURUQyT0ZlV212UjBFRHc5VWhCcmNmdXRjV3UzYWRQYjdORDB
wQWVqbERxLW5fclRDTzFzsq ~ └
&t 7https://play.google.com/store/apps/developer?
id=HaluDevsq ~ └
“t Thttp://www.myntra.com/tshirts/roadster/roadster-women-olive-
green-t-shirt/271013/buysq ~ └
=t 'http://fi.wikipedia.org/wiki/Freelancersq ~ └  *
(t +https://www.youtube.com/watch?v=9VHrC31-Lecsq ~ └  (4t
https://maktoob.yahoo.com/?
p=dnrsq ~ └  └Št 9http://www.dailyfreecode.com/Code/file-
operation-
180.aspxsq ~ └   Zt "http://stackoverflow.com/q/9394123sq ~ └  .a
t Mhttp://wwp.greenwichmeantime.com/time-zone/europe/european-
union/poland/time/sq ~ └  ,¼t :https://www.youtube.com/results?
search_query=colors&page=
4sq ~ └  'ot :https://www.youtube.com/results?
```

Fig 10: Hashmap.ser file

In the above figure we see the hashmap that is created in the Extract_Links class and stored in the hashmap.ser file.

5.3.2 Download Links

Download links is a button when clicked will download all the links that we have stored in the hashmap.ser file. In this we create the object of DeserializeFile class and then call the Read_files method. In this method it reads the hashmap.ser file and then for iterator call the URL_Download method in the URLDOWNLOAD class by passing the hashmap key to it that is the link. In the URL_Download method it reads the content of the web page nad stores it in the html file.

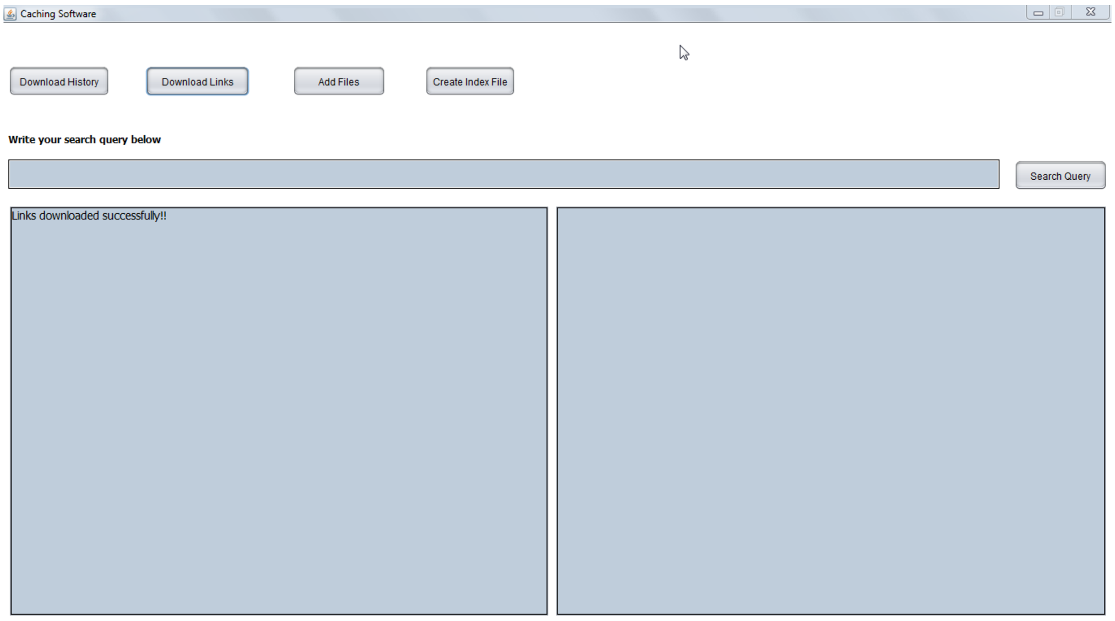
This is the code snippet for extracting the keys from the hashmap.

```
while(iterator.hasNext()) {
        Map.Entry mentry = (Map.Entry)iterator.next();
        if(mentry.getKey().toString().isEmpty())
            continue;
        else
```

```
            ur.URL_Download((String)mentry.getKey());
      }
```

This is the code for downloading the web page from the given url.

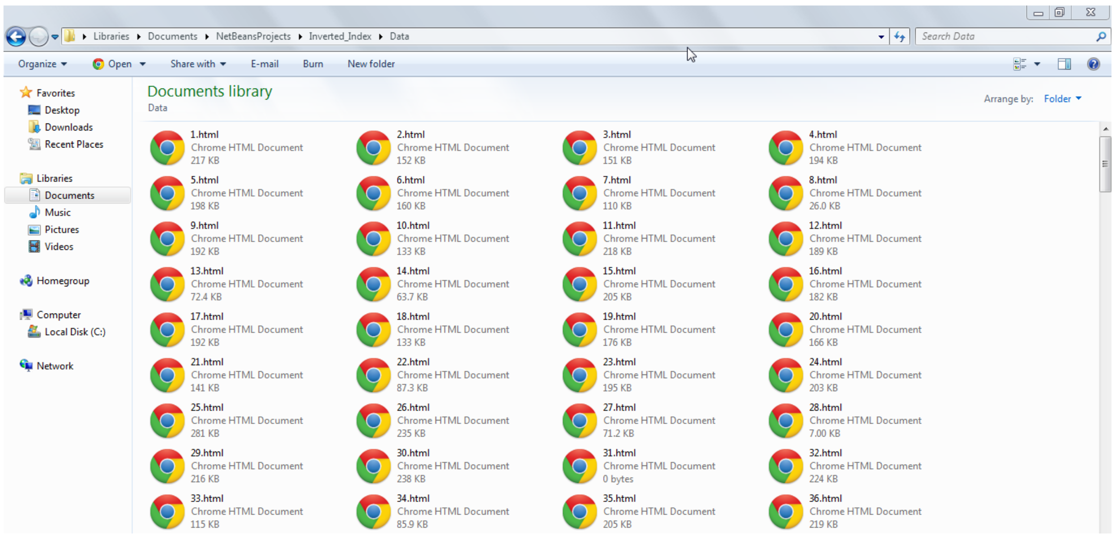
```
public static void URL_Download(String text) throws Exception
{
            URL url = new URL(text);
          BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));
      BufferedWriter writer = new
BufferedWriter(newFileWriter("C:\\Users\\RAGHAV\\Documents\\NetBeansProjects
\\Inverted_Index\\Data\\" + i +".html"));
      i++;
      String line;
     while ((line = reader.readLine()) != null)
      {
         writer.write(line);
         writer.newLine();
      }
      reader.close();
      writer.close();
    }
```



Links downloaded successfully!!

Fig 11: Download Links Button Clicked Output

From the above figure we can see that when the links that are present in the hashmap.ser file are downloaded it displays the output in the textarea saying all the links have been downloaded.



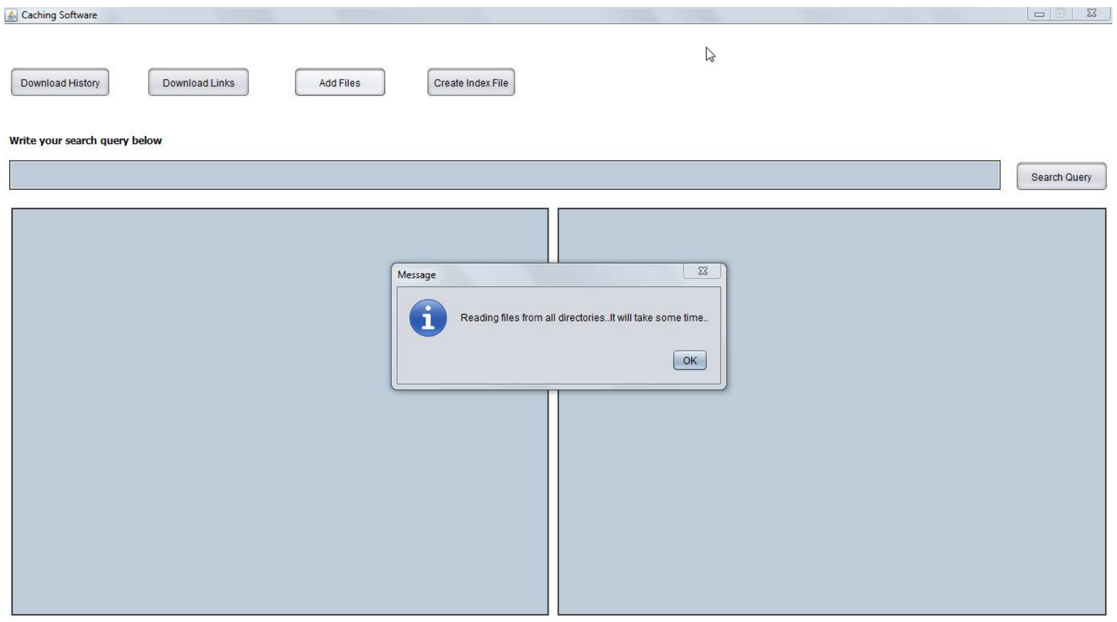
1.html Chrome HTML Document 217 KB	2.html Chrome HTML Document 152 KB	3.html Chrome HTML Document 151 KB	4.html Chrome HTML Document 194 KB
5.html Chrome HTML Document 198 KB	6.html Chrome HTML Document 160 KB	7.html Chrome HTML Document 110 KB	8.html Chrome HTML Document 26.0 KB
9.html Chrome HTML Document 192 KB	10.html Chrome HTML Document 133 KB	11.html Chrome HTML Document 218 KB	12.html Chrome HTML Document 189 KB
13.html Chrome HTML Document 72.4 KB	14.html Chrome HTML Document 63.7 KB	15.html Chrome HTML Document 205 KB	16.html Chrome HTML Document 182 KB
17.html Chrome HTML Document 192 KB	18.html Chrome HTML Document 133 KB	19.html Chrome HTML Document 176 KB	20.html Chrome HTML Document 166 KB
21.html Chrome HTML Document 141 KB	22.html Chrome HTML Document 87.3 KB	23.html Chrome HTML Document 195 KB	24.html Chrome HTML Document 203 KB
25.html Chrome HTML Document 281 KB	26.html Chrome HTML Document 235 KB	27.html Chrome HTML Document 71.2 KB	28.html Chrome HTML Document 7.00 KB
29.html Chrome HTML Document 216 KB	30.html Chrome HTML Document 238 KB	31.html Chrome HTML Document 0 bytes	32.html Chrome HTML Document 224 KB
33.html Chrome HTML Document 115 KB	34.html Chrome HTML Document 85.9 KB	35.html Chrome HTML Document 205 KB	36.html Chrome HTML Document 219 KB

Fig 12: Downloaded Documents

In the above figure we can see the html documents that are downloaded when the download links button is clicked.

5.3.3 Add Files

Add files is a button which when clicked will add all the files present under the directory and also its subdirectory. When this button is clicked then first of all the file which consists of all file names is checked whether it is empty or has some names. If it is empty then the files are added but if the file has some names then it is first asked whether you want to rewrite the file, if yes then files are added or else no changes are made. A message is popped up which displays the message that reading files from all directories. It will take some time.



Reading files from all directories..It will take some time..

Fig 13: Add Files Button Clicked Output

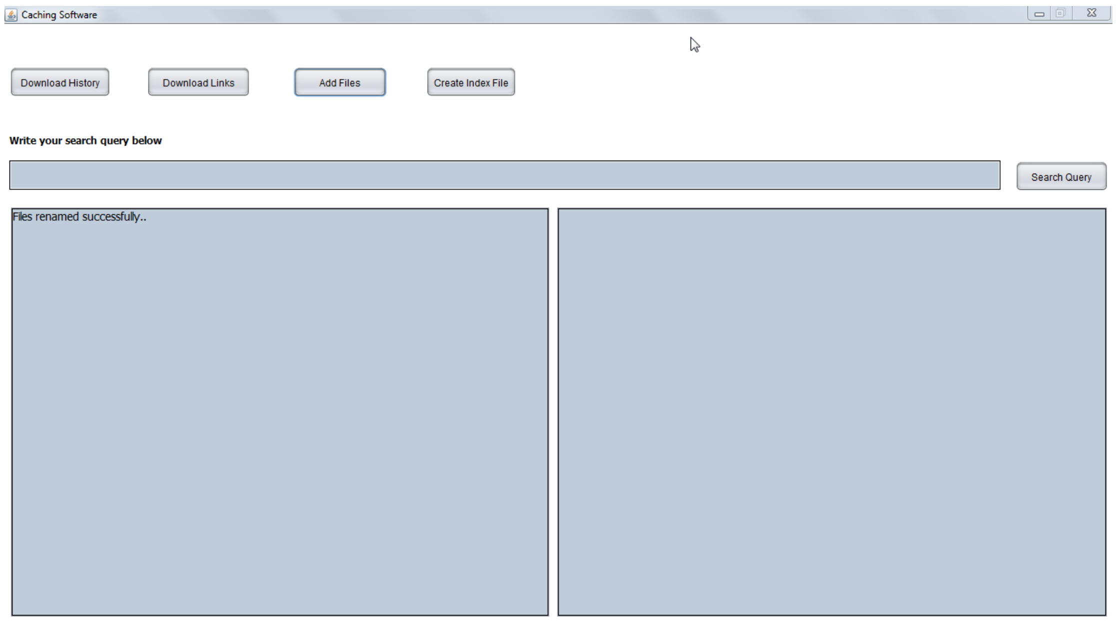
When the user clicks ok then first of all the rename function is called up and the name of the directory is passed which is fixed. When files are to be added a directory is set under which all files are present as such or under some subdirectory. From that particular subdirectory firstly all the files present under that subdirectory are listed then it is checked whether a selected item in a list is a file or a subdirectory. If it is a file then its extension is checked because currently I am working with text and pdf files only. If a file is without any extension then it is renamed to ".txt" and if the file has ".pdf" extension then first of all the content is extracted and a text file is made of same name and the pdf files is deleted from that sub-directory and copied to another folder for future use. If it is a subdirectory then the rename function is called with the current file path. This is how the files are renamed.

This is the code snippet for renaming the file.

```
public void rename(String directoryName)
   {
        boolean bool = false;
        File directory = new File(directoryName);
        File[] fList = directory.listFiles();
        String line="";
        String path =
"C:\\Users\\RAGHAV\\Documents\\NetBeansProjects\\Inverted_Index\\temp\\";
for (File file : fList)
        {
            String result="";
            if (file.isFile())
            {
                 line=file.toString();
                 String ext=getFileExtension(new File(line));
                 if(ext.equals("pdf"))
                   {
                      String line1=file.toString();
                      int i=line.indexOf("pdf");
                      result=line.substring(0,i);
                      result=result+"txt";
                      line=result;
                      String ex=line.substring(PREFACE.length(),line.length());
                      new ExtractPageContent().parsePdf(line1,line);
Files.copy(file.toPath(),(new File(path +
file.getName())).toPath(),StandardCopyOption.REPLACE_EXISTING);
file.delete();
ext="";
}
else if(ext.equals(""))
                {
                    File fnew=new File(file+ ".txt");
                    bool=file.renameTo(fnew);
            }
  }
```

This is the code snippet to check the file extension.

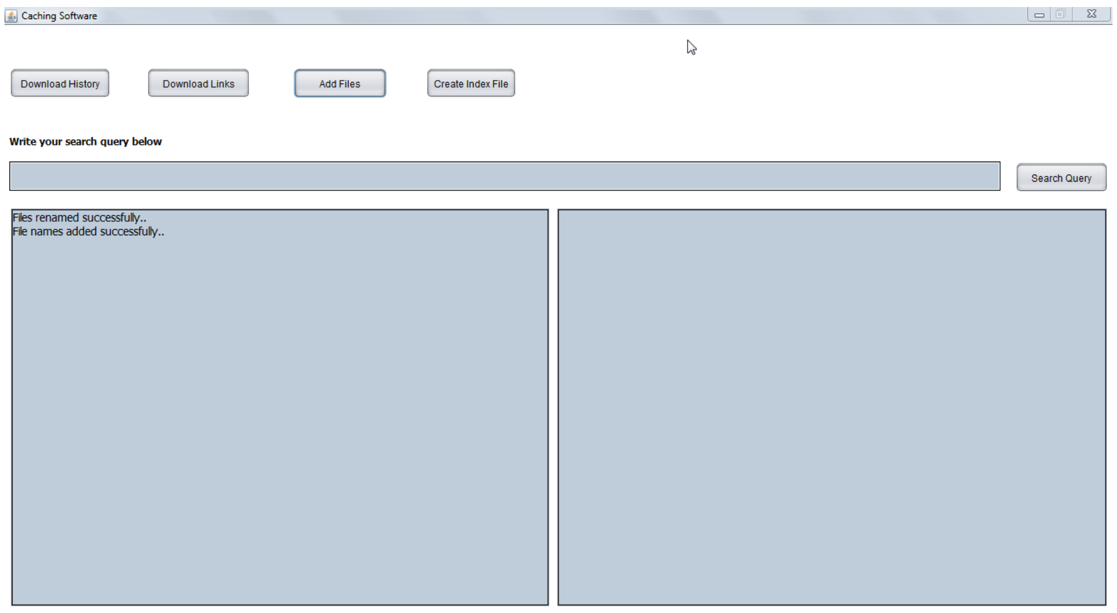
```
private static String getFileExtension(File file) {
        String fileName = file.getName();
        if(fileName.lastIndexOf(".") != -1 && fileName.lastIndexOf(".") != 0)
return fileName.substring(fileName.lastIndexOf(".")+1);
        else return "";
   }
```



```
Files renamed successfully..
```

Fig 14: Files Renamed Successfully

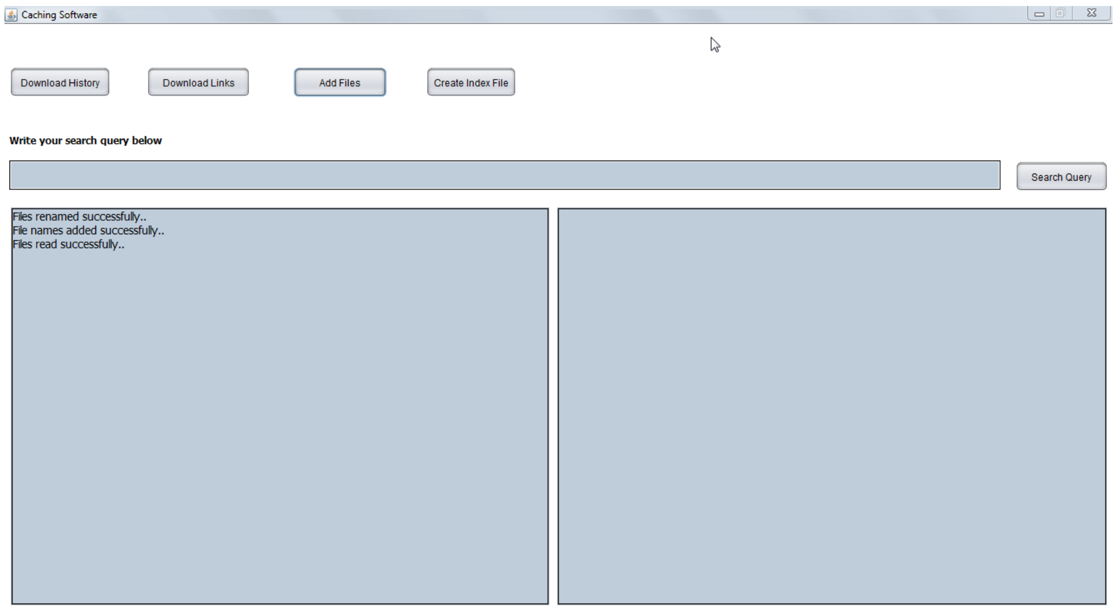
When the message appears on the text area that the files are renamed successfully, then the function listFilesAndFilesSubDirectories is called for adding all the file names to new file from where we can read the files so as for creating the inverted index. This works the same way as the rename function works but with a small change. The change is that instead of renaming the file or checking its extension this function writes the file names to the new file. A directory which consists of all the files as such or under some subdirectories is passed as an argument to this function. It also lists all the files under the directory and checks whether the selected item in a list is a file or a subdirectory. If it is a file then the file name is written to the new file else the function is called back with the current file path so as to list all files. This is how this function works. After the process is completed the text area is displayed with a message that the file names are added successfully.



Files renamed successfully..
File names added successfully..

Fig 15: Files Names Added Successfully

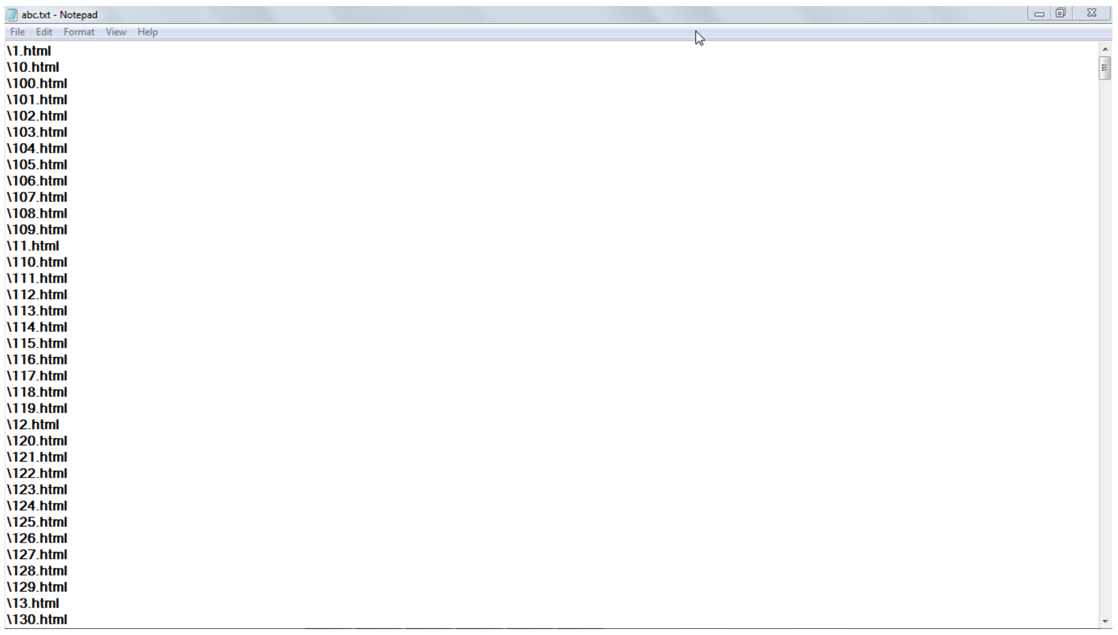
When all the functioning under the add button is completed the message is displayed in the text area telling that files are read successfully.



```
Files renamed successfully..
File names added successfully..
Files read successfully..
```

Fig 16: Files Read Successfully

This is how the file looks like when the file names are added to the new file.



```
\1.html
\10.html
\100.html
\101.html
\102.html
\103.html
\104.html
\105.html
\106.html
\107.html
\108.html
\109.html
\11.html
\110.html
\111.html
\112.html
\113.html
\114.html
\115.html
\116.html
\117.html
\118.html
\119.html
\12.html
\120.html
\121.html
\122.html
\123.html
\124.html
\125.html
\126.html
\127.html
\128.html
\129.html
\13.html
\130.html
```

Fig 17: Add Files Output

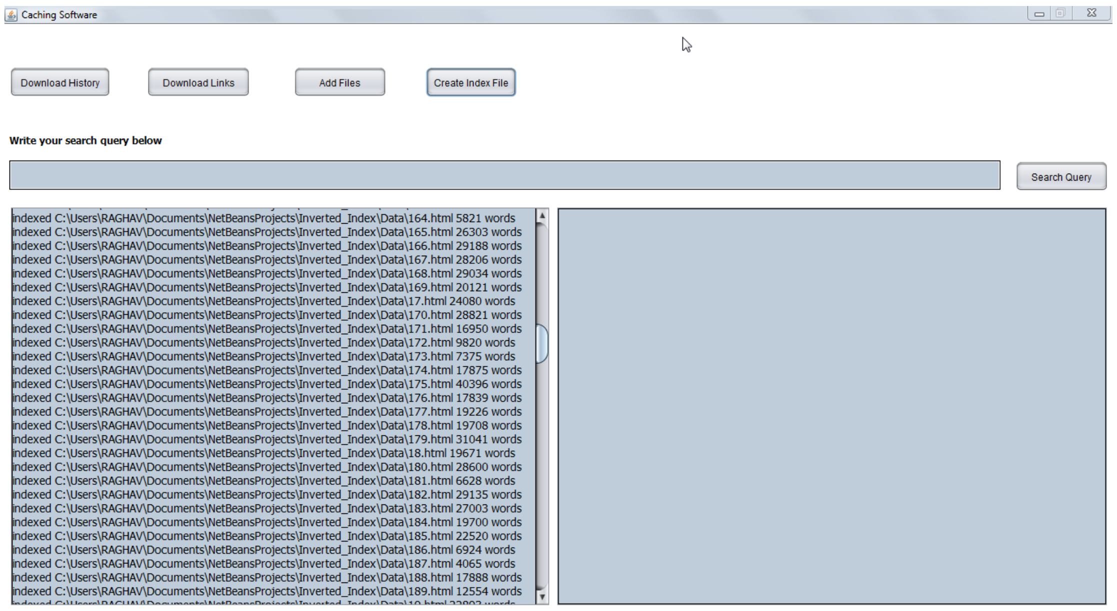
5.3.4 Create Inverted Index

After the files are added, the next step is to create the inverted index. The name of a file which consists of all the path for all the file names is set. When the create index file button is clicked first of all the line is read from the file and then the indexFile function is called up with the line as an argument.

Now first of all the file path is added and a file number is assigned to that file. For this file now we read line by line and the line is split with split function and we store that in a string and convert that string to lowercase. And we increment the count so as to count the total number of words present in the file. A stop words list is maintained so that we don't create the index for those words. A hash map is created to store the word and the id corresponding to that word. Now when we encounter our first word we check whether that word is present in the hash map or not, for this a list of type tuple is maintained which returns the result. If the id is null that means that the word is not present. Then we create a linked list of type Tuple and add the word and the Tuple to the hash map. At last we call the Tuple constructor of private class Tuple with arguments as file number and count and add that to the list of type Tuple. This way we read all the other words and index file. This is how the result is displayed after we index the file.

This is code snippet for creating the hashmap.

```
List<Tuple> idx = index.get(word);
if (idx == null)
{
     idx = new LinkedList<Tuple>();
    index.put(word, idx);
}
idx.add(new Tuple(fileno, pos));
```



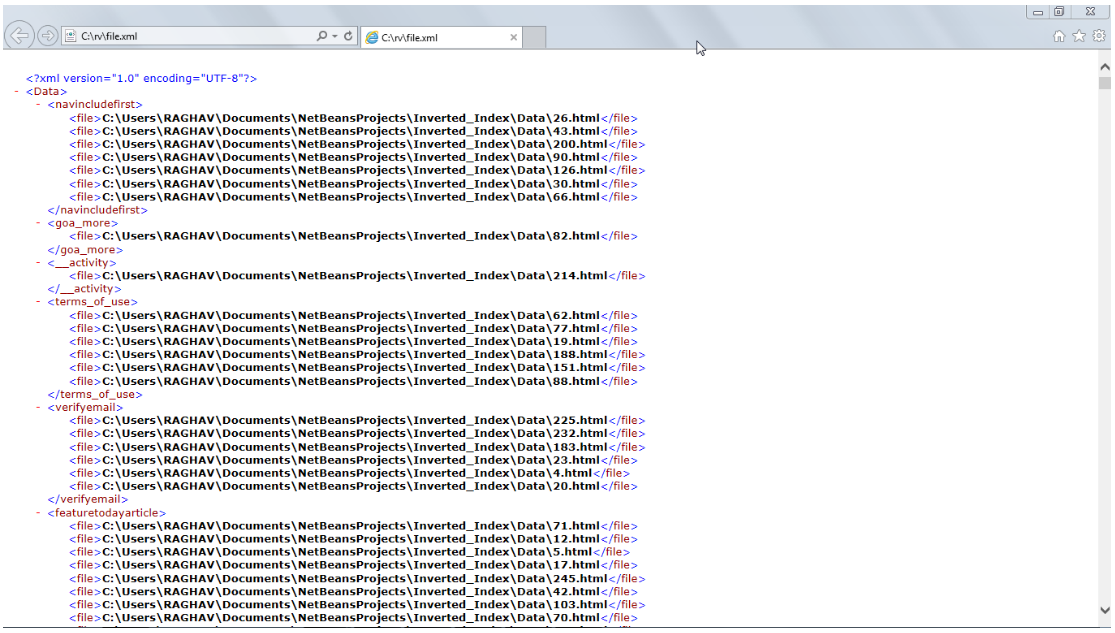
```
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\164.html 5821 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\165.html 26303 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\166.html 29188 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\167.html 28206 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\168.html 29034 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\169.html 20121 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\17.html 24080 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\170.html 28821 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\171.html 16950 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\172.html 9820 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\173.html 7375 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\174.html 17875 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\175.html 40396 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\176.html 17839 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\177.html 19226 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\178.html 19708 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\179.html 31041 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\18.html 19671 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\180.html 28600 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\181.html 6628 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\182.html 29135 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\183.html 27003 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\184.html 19700 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\185.html 22520 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\186.html 6924 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\187.html 4065 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\188.html 17888 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\189.html 12554 words
```

Fig 18: Files Indexed Successfully Output

After the indexing is done next step is to create the inverted index for all the words present in the hash map. For this we create an array list of type string and add all the key set present in hash map to this list. And we sort all the list so as to create the sorted inverted index of words. Now we check the Tuple for that particular word. If the tuple is present then we retrieve the filename and store that in a hash set and then write to the index file. This way we check for all the array list and create the inverted index for those words. After creating the inverted index this is how the file looks like.

This is code snippet for creating the xml file.

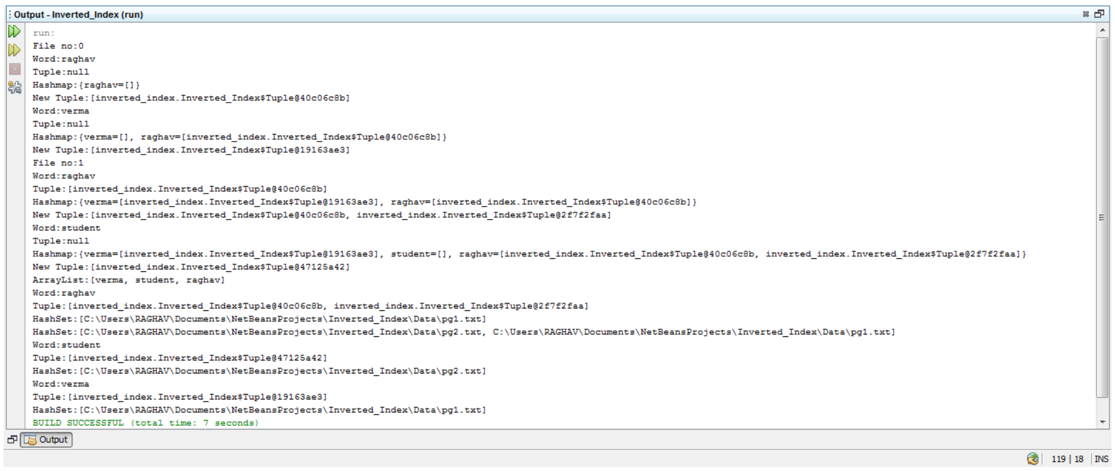
```
DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
        Document doc = docBuilder.newDocument();
        Element rootElement = doc.createElement("Data");
```



```
<?xml version="1.0" encoding="UTF-8"?>
- <Data>
  - <navincludefirst>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\26.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\43.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\200.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\90.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\126.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\30.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\66.html</file>
    </navincludefirst>
  - <goa_more>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\82.html</file>
    </goa_more>
  - <__activity>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\214.html</file>
    </__activity>
  - <terms_of_use>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\62.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\77.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\19.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\188.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\151.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\88.html</file>
    </terms_of_use>
  - <verifyemail>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\225.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\232.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\183.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\23.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\4.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\20.html</file>
    </verifyemail>
  - <featuretodayarticle>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\71.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\12.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\5.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\17.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\245.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\42.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\103.html</file>
      <file>C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\70.html</file>
```

Fig 19: Indexing Files Output

Small demonstration of the output when we are indexing file and making the inverted index. I have two files under the fixed directory. The file names are pg1.txt and pg2.txt. pg1.txt has words {raghav,verma} and pg2.txt has words {raghav,student}. This is how the indexing of a file and creating inverted index looks like.

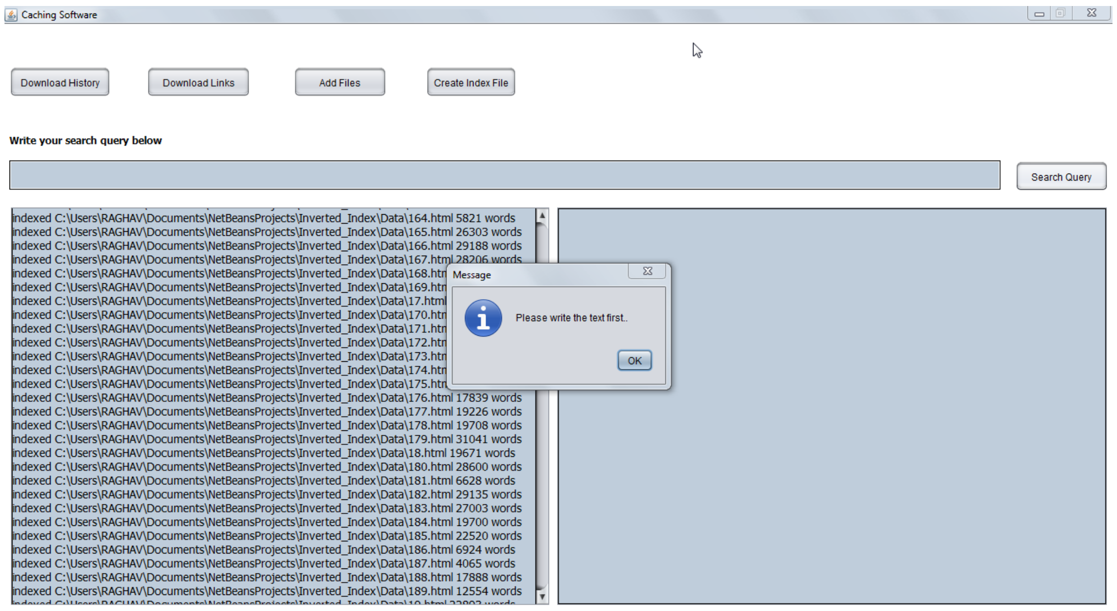


```
run:
File no:0
Word:raghav
Tuple:null
Hashmap:{raghav=[]}
New Tuple:[inverted_index.Inverted_Index$Tuple@40c06c8b]
Word:verma
Tuple:null
Hashmap:{verma=[], raghav=[inverted_index.Inverted_Index$Tuple@40c06c8b]}
New Tuple:[inverted_index.Inverted_Index$Tuple@19163ae3]
File no:1
Word:raghav
Tuple:[inverted_index.Inverted_Index$Tuple@40c06c8b]
Hashmap:{verma=[inverted_index.Inverted_Index$Tuple@19163ae3], raghav=[inverted_index.Inverted_Index$Tuple@40c06c8b]}
New Tuple:[inverted_index.Inverted_Index$Tuple@40c06c8b, inverted_index.Inverted_Index$Tuple@2f7f2faa]
Word:student
Tuple:null
Hashmap:{verma=[inverted_index.Inverted_Index$Tuple@19163ae3], student=[], raghav=[inverted_index.Inverted_Index$Tuple@40c06c8b, inverted_index.Inverted_Index$Tuple@2f7f2faa]}
New Tuple:[inverted_index.Inverted_Index$Tuple@47125a42]
ArrayList:[verma, student, raghav]
Word:raghav
Tuple:[inverted_index.Inverted_Index$Tuple@40c06c8b, inverted_index.Inverted_Index$Tuple@2f7f2faa]
HashSet:[C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\pg1.txt]
HashSet:[C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\pg2.txt, C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\pg1.txt]
Word:student
Tuple:[inverted_index.Inverted_Index$Tuple@47125a42]
HashSet:[C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\pg2.txt]
Word:verma
Tuple:[inverted_index.Inverted_Index$Tuple@19163ae3]
HashSet:[C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\pg1.txt]
BUILD SUCCESSFUL (total time: 7 seconds)
```

Fig 20: Example of Inverted Index

5.3.5 Searching

After adding the files and creating the inverted index file, the last option left is to search for a query. When the search query button is clicked first the string is checked whether it is empty or not. If a string is empty a message is displayed saying please write the text first, else a normal search operation is performed.



Please write the text first..

```
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\164.html 5821 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\165.html 26303 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\166.html 29188 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\167.html 28206 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\168.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\169.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\17.html
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\170.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\171.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\172.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\173.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\174.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\175.htm
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\176.html 17839 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\177.html 19226 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\178.html 19708 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\179.html 31041 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\18.html 19671 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\180.html 28600 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\181.html 6628 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\182.html 29135 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\183.html 27003 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\184.html 19700 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\185.html 22520 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\186.html 6924 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\187.html 4065 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\188.html 17888 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\189.html 12554 words
```

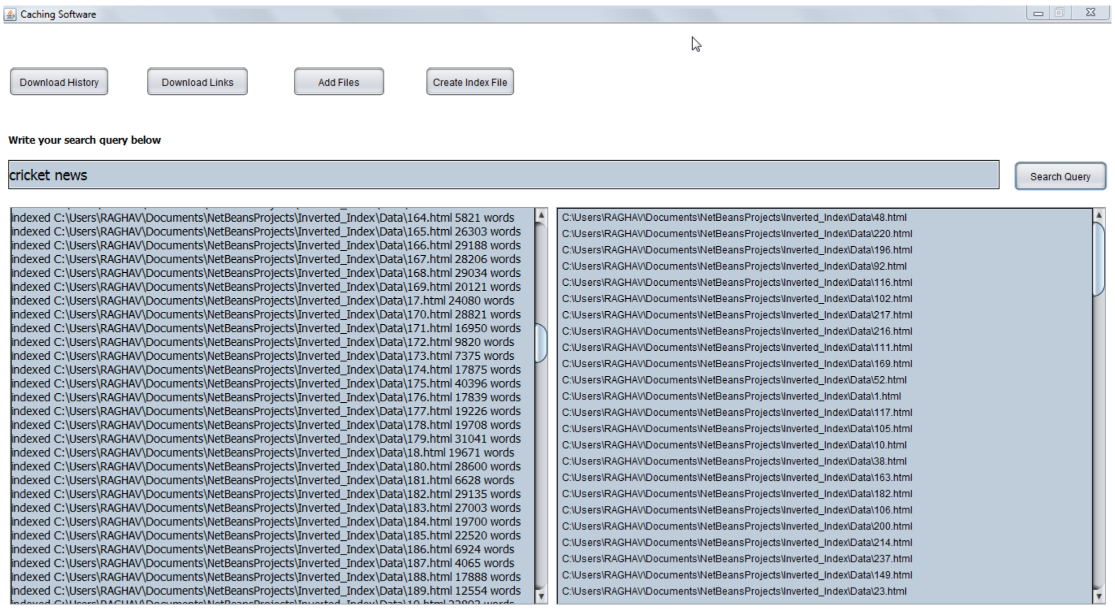
Fig 21: Search Button Clicked Message

Now when the text is passed firstly all the special characters and white spaces are removed. The string is split into individual word and appended with a comma after each word. When the search function in the inverted class is called the string is split with comma and passed as a list of strings.

When the search function receives arguments as a list of strings, each word is checked in the hashmap which was created while we were creating the inverted index. We have created the arraylist of arraylist. For each word present in the search query we check the hashmap whether it contains the word or not. If the word is present in the hashmap then we create the new arraylist of type string and we add the file names to the arraylist. Now to find the intersection between the files simply check the filename in the list, if it is present then we add that filename to the final list and then display the output.

This is code snippet for creating the arraylist of arraylist of type string.

```
for (String _word : wordsn)
         {
             if(words.containsKey(_word))
             {
                 bogus.add(new ArrayList<String>());
words.get(_word).toString().replace("[", "").replace("]", "").trim();
for(String _list : words.get(_word))
                 {
                     bogus.get(i).add(_list);
                 }
                     i++;
             }
         }
```



cricket news

```
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\164.html 5821 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\165.html 26303 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\166.html 29188 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\167.html 28206 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\168.html 29034 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\169.html 20121 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\17.html 24080 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\170.html 28821 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\171.html 16950 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\172.html 9820 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\173.html 7375 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\174.html 17875 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\175.html 40396 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\176.html 17839 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\177.html 19226 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\178.html 19708 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\179.html 31041 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\18.html 19671 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\180.html 28600 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\181.html 6628 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\182.html 29135 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\183.html 27003 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\184.html 19700 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\185.html 22520 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\186.html 6924 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\187.html 4065 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\188.html 17888 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\189.html 12554 words
```

```
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\48.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\220.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\196.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\92.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\116.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\102.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\217.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\216.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\111.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\169.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\52.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\1.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\117.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\105.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\10.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\38.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\163.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\182.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\106.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\200.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\214.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\237.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\149.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\23.html
```

Fig 22: Search Result Output

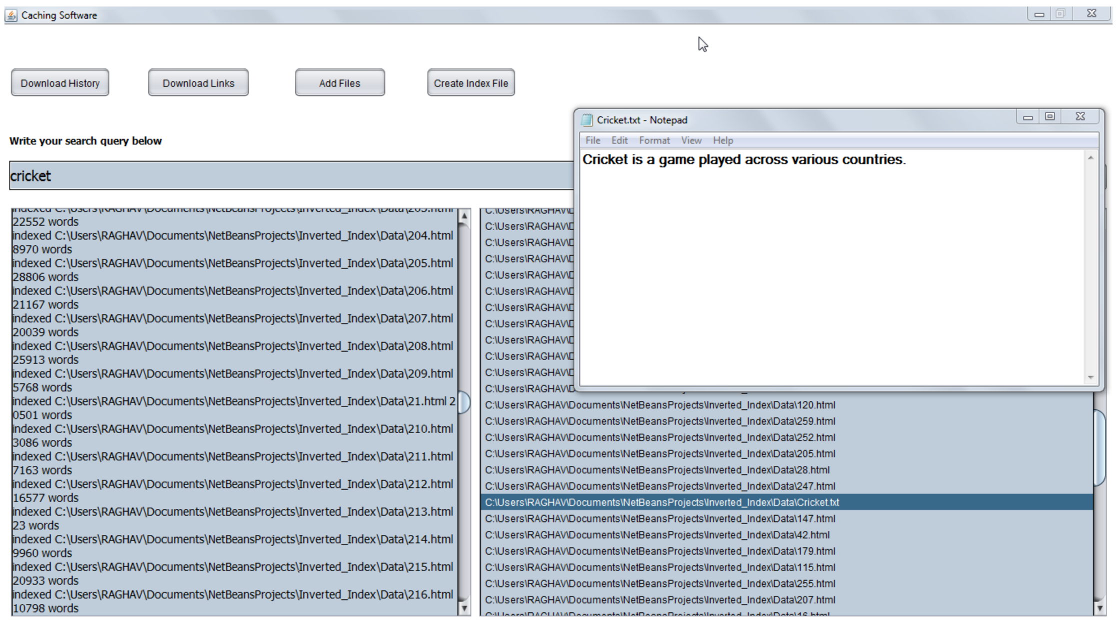
This is code snippet for taking the intersection between the files.

```
while(index_num<bogus.size())
             {
                 for(String t:bogus.get(0))
                 {
                      if(final_list.contains(t))
```

```
                continue;
            else
             {

                boolean checkb=temp_list.contains(t);
                if(checkb)
                    continue;
                else if(bogus.get(index_num).contains(t))
                       final_list.add(t);
                else
                    temp_list.add(t);
             }
        }
        index_num++;
    }
```

Now I have added the functionality that when the output is displayed in the jlist then the user can click on any filename and open the file from there itself. As we can see in the below figure when we click on the cricket.txt then the files open up.



```
cricket
```

```
22552 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\204.html
8970 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\205.html
28806 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\206.html
21167 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\207.html
20039 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\208.html
25913 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\209.html
5768 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\21.html 2
0501 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\210.html
3086 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\211.html
7163 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\212.html
16577 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\213.html
23 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\214.html
9960 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\215.html
20933 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\216.html
10798 words
```

```
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\D
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\120.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\259.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\252.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\205.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\28.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\247.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\Cricket.txt
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\147.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\42.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\179.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\115.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\255.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\207.html
```

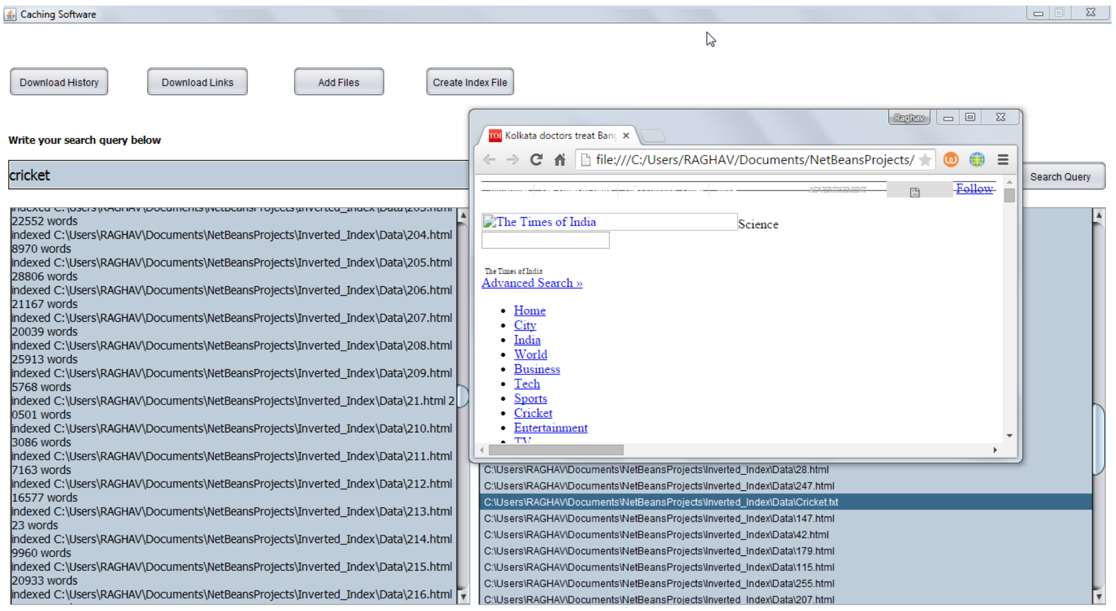
```
Cricket is a game played across various countries.
```

Fig 23: Text File Clicked Output

We can see in the below figure that when we click on the file with the .html extension the file opens up in the browser and user can look up whatever he wants to see in that page.

This is code snippet for opening up a file when clicked on the file name.

```
File htmlFile = new File(selected);
Desktop.getDesktop().browse(htmlFile.toURI());
```



```
22552 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\204.html
8970 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\205.html
28806 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\206.html
21167 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\207.html
20039 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\208.html
25913 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\209.html
5768 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\21.html 2
0501 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\210.html
3086 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\211.html
7163 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\212.html
16577 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\213.html
23 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\214.html
9960 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\215.html
20933 words
indexed C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\216.html
```

```
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\28.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\247.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\Cricket.txt
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\147.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\42.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\179.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\115.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\255.html
C:\Users\RAGHAV\Documents\NetBeansProjects\Inverted_Index\Data\207.html
```

Fig 24: HTML File Clicked Output

Chapter 6: Ranking

Ranking of query results is one of the fundamental problems in information retrieval (IR), the scientific/engineering discipline behind search engines. Given a query *q* and a collection *D* of documents that match the query, the problem is to rank, that is, sort, the documents in *D* according to some criterion so that the "best" results appear early in the result list displayed to the user. Classically, ranking criteria are phrased in terms of relevance of documents with respect to an information need expressed in the query.

Ranking is often reduced to the computation of numeric scores on query/document pairs; a baseline score function for this purpose is the cosine similarity between tf–idf vectors representing the query and the document in a vector space model, BM25 scores, or probabilities in a probabilistic IR model. A ranking can then be computed by sorting documents by descending score. An alternative approach is to define a score function on pairs of documents d_1, d_2 that is positive if and only if d_1 is more relevant to the query than d_2 and using this information to sort.

Ranking functions are evaluated by a variety of means; one of the simplest is determining the precision of the first *k* top-ranked results for some fixed *k*; for example, the proportion of the top 10 results that are relevant, on average over many queries.

Frequently, computation of ranking functions can be simplified by taking advantage of the observation that only the relative order of scores matters, not their absolute value; hence terms or factors that are independent of the document may be removed, and terms or factors that are independent of the query may be precomputed and stored with the document.

6.1 Hits Algorithm

Hyperlink-Induced Topic Search (HITS; also known as hubs and authorities) is a link analysis algorithm that rates Web pages, developed by Jon Kleinberg. The idea behind Hubs and Authorities stemmed from a particular insight into the creation of web pages when the Internet was originally forming; that is, certain web pages, known as hubs, served as large directories that were not actually authoritative in the information that it held, but were used as compilations of a broad catalog of information that led users directly to other authoritative pages. In other words, a good hub represented a page that pointed to many other pages, and a good authority represented a page that was linked by many different hubs.

The scheme therefore assigns two scores for each page: its authority, which estimates the value of the content of the page, and its hub value, which estimates the value of its links to other pages.

6.1.1 Algorithm

In the HITS algorithm, the first step is to retrieve the most relevant pages to the search query. This set is called the root set and can be obtained by taking the top n pages returned by a text-based search algorithm. A base set is generated by augmenting the root set with all the web pages that are linked from it and some of the pages that link to it. The web pages in the base set and all hyperlinks among those pages form a focused sub graph. The HITS computation is performed only on this focused sub graph. According to Kleinberg the reason for constructing a base set is to ensure that most (or many) of the strongest authorities are included.

Authority and hub values are defined in terms of one another in a mutual recursion. An authority value is computed as the sum of the scaled hub values that point to that page. A hub value is the sum of the scaled authority values of the pages it points to. Some implementations also consider the relevance of the linked pages.

The algorithm performs a series of iterations, each consisting of two basic steps:

- **Authority Update**: Update each node's Authority score to be equal to the sum of the Hub Scores of each node that points to it. That is, a node is given a high authority score by being linked from pages that are recognized as Hubs for information.
- **Hub Update**: Update each node's Hub Score to be equal to the sum of the Authority Scores of each node that it points to. That is, a node is given a high hub score by linking to nodes that are considered to be authorities on the subject.

The Hub score and Authority score for a node is calculated with the following algorithm:

- Start with each node having a hub score and authority score of 1.
- Run the Authority Update Rule
- Run the Hub Update Rule
- Normalize the values by dividing each Hub score by square root of the sum of the squares of all Hub scores, and dividing each Authority score by square root of the sum of the squares of all Authority scores.
- Repeat from the second step as necessary.

HITS, like Page and Brin's PageRank, is an iterative algorithm based on the linkage of the documents on the web. However it does have some major differences:

- It is query dependent, that is, the (Hubs and Authority) scores resulting from the link analysis are influenced by the search terms;
- As a corollary, it is executed at query time, not at indexing time, with the associated hit on performance that accompanies query-time processing.
- It is not commonly used by search engines.
- It computes two scores per document, hub and authority, as opposed to a single score;
- It is processed on a small subset of ‘relevant’ documents (a 'focused sub graph' or base set), not all documents as was the case with PageRank.

6.1.2 Detail Explanation

To begin the ranking, $\forall p$, $\mathrm{auth}(p) = 1$ and $\mathrm{hub}(p) = 1$. We consider two types of updates: Authority Update Rule and Hub Update Rule. In order to calculate the hub/authority scores of each node, repeated iterations of the Authority Update Rule and the Hub Update Rule are applied. A k-step application of the Hub-Authority algorithm entails applying for k times first the Authority Update Rule and then the Hub Update Rule.

Authority Update Rule

$\forall p$, we update $\mathrm{auth}(p)$ to be the summation:

$$\mathrm{auth}(p) = \sum_{i=1}^{n} \mathrm{hub}(i)$$

where n is the total number of pages connected to p and i is a page connected to p. That is, the Authority score of a page is the sum of all the Hub scores of pages that point to it.

Hub Update Rule

$\forall p$, we update $\mathrm{hub}(p)$ to be the summation:

$$\mathrm{hub}(p) = \sum_{i=1}^{n} \mathrm{auth}(i)$$

where n is the total number of pages p connects to and i is a page which p connects to. Thus a page's Hub score is the sum of the Authority scores of all its linking pages

Normalization

The final hub-authority scores of nodes are determined after infinite repetitions of the algorithm. As directly and iteratively applying the Hub Update Rule and Authority Update Rule leads to diverging values, it is necessary to normalize the matrix after every iteration. Thus the values obtained from this process will eventually converge.

Pseudocode

```
 G := set of pages
 for each page p in G do
   p.auth = 1 // p.auth is the authority score of the page p
   p.hub = 1 // p.hub is the hub score of the page p
 function HubsAndAuthorities(G)
   for step from 1 to k do // run the algorithm for k steps
     norm = 0
```

```
  for each page p in G do  // update all authority values first
    p.auth = 0
    for each page q in p.incomingNeighbors do // p.incomingNeighbors is the set of pages that link to p
      p.auth += q.hub
    norm += square(p.auth) // calculate the sum of the squared auth values to normalise
  norm = sqrt(norm)
  for each page p in G do  // update the auth scores
    p.auth = p.auth / norm  // normalise the auth values
  norm = 0
  for each page p in G do  // then update all hub values
    p.hub = 0
    for each page r in p.outgoingNeighbors do // p.outgoingNeighbors is the set of pages that p links to
      p.hub += r.auth
    norm += square(p.hub) // calculate the sum of the squared hub values to normalise
  norm = sqrt(norm)
  for each page p in G do  // then update all hub values
    p.hub = p.hub / norm   // normalise the hub values
```

The hub and authority values converge in the pseudocode above.

The code below does not converge, because it is necessary to limit the number of steps that the algorithm runs for. One way to get around this, however, would be to normalize the hub and authority values after each "step" by dividing each authority value by the square root of the sum of the squares of all authority values, and dividing each hub value by the square root of the sum of the squares of all hub values. This is what the pseudocode above does.

Non-converging pseudocode

```
G := set of pages
 for each page p in G do
   p.auth = 1 // p.auth is the authority score of the page p
   p.hub = 1 // p.hub is the hub score of the page p
 function HubsAndAuthorities(G)
   for step from 1 to k do // run the algorithm for k steps
     for each page p in G do  // update all authority values first
       p.auth = 0
       for each page q in p.incomingNeighbors do // p.incomingNeighbors is the set of pages that link to p
         p.auth += q.hub
     for each page p in G do  // then update all hub values
       p.hub = 0
       for each page r in p.outgoingNeighbors do // p.outgoingNeighbors is the set of pages that p links to
         p.hub += r.auth
```

6.2 Page Rank

The citation (link) graph of the web is an important resource that has largely gone unused in existing web search engines. They have created maps containing as many as 518 million of these hyperlinks, a significant sample of the total. These maps allow rapid calculation of a web page’s “PageRank”, an objective measure of its citation importance that corresponds well with people’s subjective idea of importance.

Because of this correspondence, PageRank is an excellent way to prioritize the results of web keyword searches. For most popular subjects, a simple text matching search that is restricted to web page titles performs admirably when PageRank prioritizes the results (demo available at google.standford.edu). For the type of full text searches in the main Google system, PageRank also helps a great deal.

6.2.1 Description of PageRank Calculation

PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iteration”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

A probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a “50% chance” of something happening. Hence, a PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank.

Simplified Algorithm

Assume a small universe of four web pages: A, B, C and D. Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial PageRank of 1. However, later versions of PageRank, and the remainder of this section assume a probability distribution between 0 and 1. Hence the initial value for each page is 0.25.

The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.

If the only links in the system were from pages B, C and D to A, each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.

PR(A)=PR(B)+PR(C)+PR(D)

Suppose instead that page B had a link to pages C and A, while page D had links to all three pages. Thus, upon the next Iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C.

Since D had three outbound links, it would transfer one third of its existing value, or approximately 0.083, to A.

$$PR\ (A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the number of outbound links L().

$$PR\ (A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}$$

In the general case, the PageRank value for any page u can be expressed as:

$$PR\ (u) = \sum_{v€\mathrm{Bu}}^{n} \frac{PR(V)}{L(V)}$$

i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set B_u (the set containing all pages linking to page u), divided by the number L(v) of links from page v.

Damping Factor

The PageRank theory holds that even an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor d. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85.

The damping factor is subtracted from 1 (and in some variations of the algorithm, the result is divided by the number of documents (N) in the collection) and this term is then added to the product of the damping factor and the sum of the incoming PageRank scores. That is,

PR (A) = $\frac{1-d}{N} + d(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots)$

So any page's PageRank is derived in large part from the PageRank of other pages. The damping factor adjusts the derived value downward. The formula, which has led to following confusion:

PR (A) = $1 - d + d(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots)$

6.2.2 Intuitive Justification

PageRank can be thought of as a model of user behaviour. We assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank. And, the d damping factor is probability at each page the "random surfer" will get bored and request another random page. One important variation is to only add the damping factor d to a single page, or a group of pages. This allows for personalization and can make it nearly impossible to deliberately mislead the system in order to get a higher ranking. We have several other extensions to PageRank.

Another intuitive justification is that a page can have a high PageRank if there are many pages that point to it, or if there are some pages that point to it and have a high PageRank. Intuitively, pages that are well cited from many places around the web are worth looking at. Also, pages that have perhaps only one citation from something like the Yahoo! Homepage are also generally worth looking at. If a page was not high quality, or was a broken link, it is quite likely that Yahoo's homepage would not link to it. PageRank handles both these cases and everything in between by recursively propagating weights through the link structure of the web.

6.2.3 Anchor Text

The text of links is treated in a special way in our search engine. Most search engines associate the text of a link with the page that the link is on. In addition, we associate it with the page the link points to. This has several advantages. First, anchors often provide more accurate descriptions of web pages than the pages themselves. Second, anchors may exist for documents which cannot be indexed by a text-based search engine, such as images, programs, and databases.

This makes it possible to return web pages which have not actually been crawled. Note that pages that have not been crawled can cause problems, since they are never checked for validity before being returned to the user. In this case, the search engine can even return a page that never actually existed, but had hyperlinks pointing to it. However, it is possible to sort the results, so that this particular problem rarely happens. This idea of propagating anchor text to the page it refers to was implemented in the World Wide Web Worm especially because it helps search non-text information, and expands the search coverage with fewer downloaded documents. We use anchor propagation mostly because anchor text can help provide better quality results. Using anchor text efficiently is technically difficult because of the large amounts of data which must be processed. In our current crawl of 24 million pages, we had over 259 million anchors which we indexed.

6.2.4 The Ranking System

Google maintains much more information about web documents than typical search engines. Every hit list includes position, font and capitalization information.

Additionally, we factor in hits from anchor text and the PageRank of the document. Combining all of this information into a rank is difficult. We designed our ranking function so that no particular factor can have too much influence. First, consider the simplest case – a single word query. In order to rank a document with a single word query, Google looks at that document’s hit list for that word. Google considers each hit to be one of several different types (title, anchor, URL, plain text large font, plain text small font, etc...) each of which has its own type-weight.

The type weights make up a vector indexed by type. Google counts the number of hits of each type in the hit list. Then every count is converted into a count-weight. Count-weights increase linearly with counts at first but quickly taper off so that more than a certain count will not help. We take the dot product of the vector of count-weights with the vector of type-weights to compute an IR score for the document. Finally, the IR score is combined with PageRank to give a final rank to the document. For a multi word search, the situation is more complicated.

Now multiple hit lists must be scanned through at once so that hits occurring close together in a document are weighted higher than hits occurring far apart. The hits from the multiple hit lists are matched up so that nearby hits are matched together. For every matched set of hits, proximity is computed. The proximity is based on how far apart the hits are in the document (or anchor) but is classified into 10 different value “bins” ranging from a phrase match to “not even close”.

Counts are computed not only for every type of hit but for every type and proximity. Every type and proximity pair has a type-prox-weight. The counts are converted into count-weights and we take the dot product of the count-weights and the type-prox-weights to compute an IR score. All of these numbers and matrices can all be displayed with the search results using a special debug mode. These displays have been very useful in developing the ranking system.

Chapter 7: Conclusion and Future Work

The application building part of the overall application has come to a point, where I have made the program for downloading history, downloading links, adding files, creating inverted index and perform the search operation.

The final Caching Software is an application where a user will search for any data and the software will check for the data in its data repository, if it founds the equivalent data then the corresponding results will be displayed else the browser will search for the data and download the files for the corresponding data and display the output to the user. The application also has an option where the history is provided and then the links in those pages are fetched. Those links are fetched from the file and then those links are downloaded.

The future work of the application will make the application to index other documents like doc file, pdf file, xls file and more.

References

1. Anita Madaan, N. M. (2012). Performance Analysis of Web Caching Through Cache Replacement Based on User Behavior. *International Journal of Advanced Research in Computer Science and Software Engineering*, pp. 9.
2. Davison, B. (2001). A web caching primer. *IEEE*, pp. 8.
3. JIANLIANG XU, J. L. (2004). Caching and Prefetching for web content distribution. *IEEE*, pp. 6.
4. Kin Yeung Wong, K. H. (2001). Site-based approach to Web cache design. *IEEE*, pp. 7.
5. Nottingham, M. (n.d.). *Caching Tutorial for Web Authors and Webmasters*. Retrieved from mnot: https://www.mnot.net/cache_docs/
6. Page, S. B. (1998). *The Anatomy of a Large-Scale hypertextual web search engine*. Retrieved from Infolab: infolab.stanford.edu/~backrub/google.html
7. Sathiyamoorthi, T. S. (2013). Survey on Integrating Web Caching and Pre-Fetching. *International Journal of Advanced Research in Computer Science and Software Engineering*, pp. 5.