EMERTXE TRAINING PROJECT DOCUMENTATION FRAMEWORK

REQUIREMENTS & DESIGN DOCUMENT

Emertxe Information Technologies (P) Ltd

Text Indexer

VERSION: 0.1 ***REVISION DATE: 18.04.2017***

Version	Date	Changed By	Modifications
0.1	*18-04-2017*	*Satyanarayana S*	*Initial Draft*

Search Inverted Files

Table of Contents

Search Inverted Files....1
Introduction....1
Description....1
Forward Indexing....1
Inverted Indices....1
Requirement Details....3
Indexing....3
Searching....4
Module Level 5
Applications....7
References....8

Introduction

An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database. The inverted file may be the database file itself, rather than its index. It is the most popular data structure used in document retrieval systems, used on a large scale for example in search engines.

Description

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.

Forward Indexing

Creating the initial word list requires several different operations. First, the individual words must be recognized from the text. Then these words can be stored using a dynamic Linked List or Hashing. Along with the words the information about the corresponding document or file should also be stored . Like that all documents in the database are indexed. Words which are repeated in different or same files are not indexed separately, but the same word is updated with a list of those file names, in which they occur. So each word in the list is mapped with all the files in which it contain. The detail of the files can be stored as URL or as names etc.

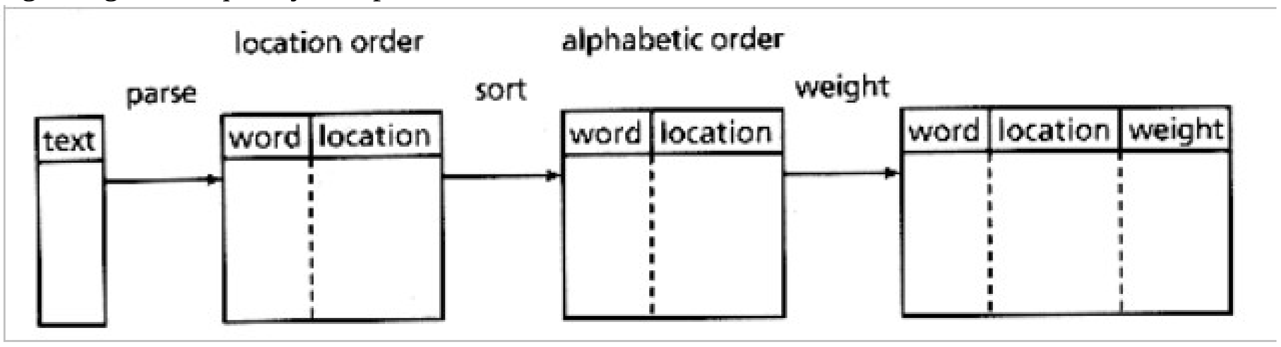
Inverted Indices

The forward index is sorted to transform it to an inverted index. This can be done with the help of a best sorting method. The list is arranged in sorted order of words. This will help to search the words easily in the list and produce the information about the documents in which they are present.

Most computers cannot sort the very large disk files needed to hold the initial word list within a reasonable time frame, and do not have the amount of storage necessary to hold a sorted and unsorted version of that word list, plus the intermediate files involved in the internal sort.

This can be avoided by implementing Hashing or sorted LinkedList or Binary Search tree at the time of Indexing. While Indexing, before storing,the words are compared and arranged in the sorting order. This helps to reduce the time complexity in searching for a word through the list.

The efficiency can be increased by implementing Hashing, which stores words based on unique indices. So while retrieval of words also the efficiency can be obtained. This index can only determine whether a word exists within a particular document, since it stores no information regarding the frequency and position of the word; it is therefore considered to be a boolean index.



Such an index determines which documents match a query but does not rank matched documents. In some designs the index includes additional information such as the frequency of each word in each document or the positions of a word in each document. Position information enables the search algorithm to identify word proximity to support searching for phrases; frequency can be used to help in ranking the relevance of documents to the query. Such topics are the central research focus of information retrieval.

Requirement Details

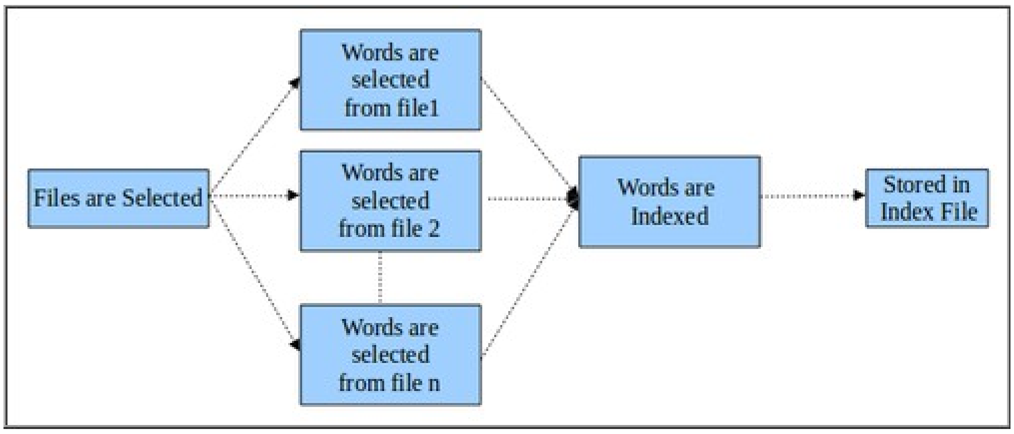
Implementing this search program mainly consits two important functions.

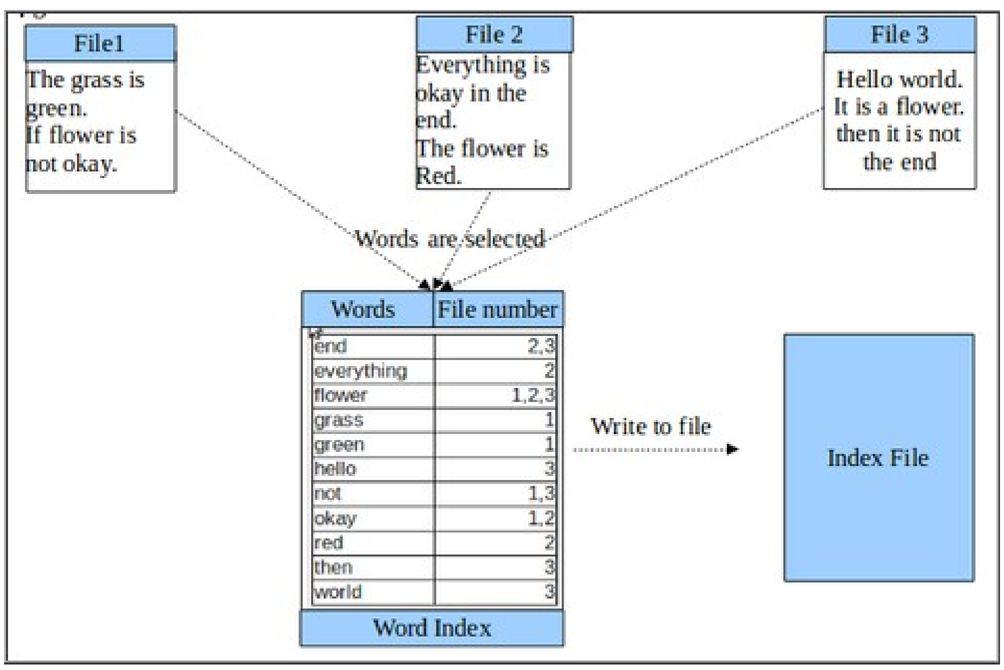
1. Indexing

2. Querying.

Indexing

By Indexing, we are creating a database file which contains the index of all words. So this can be termed as Database Creation also. All the files whose index are to be created are selected and inputed to this function. All the files are parsed and words are separated and indexed. They are arranged in sorted order. For this a sorted Linked List or Hashing is used which will store the words and the related file details. The index thus created is then stored in the file as database. This file is later used in Querying. While the files are removed or added this index file is updated.


Files are Selected
Words are selected from file1
Words are selected from file 2
Words are selected from file n
Words are Indexed
Stored in Index File



File1: The grass is green. If flower is not okay.

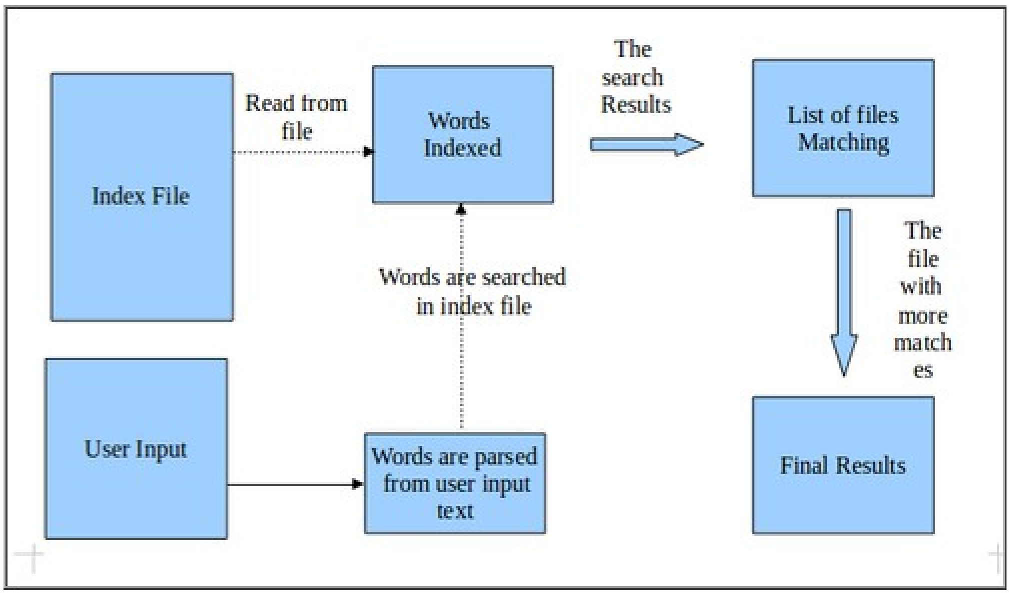
File 2: Everything is okay in the end. The flower is Red.

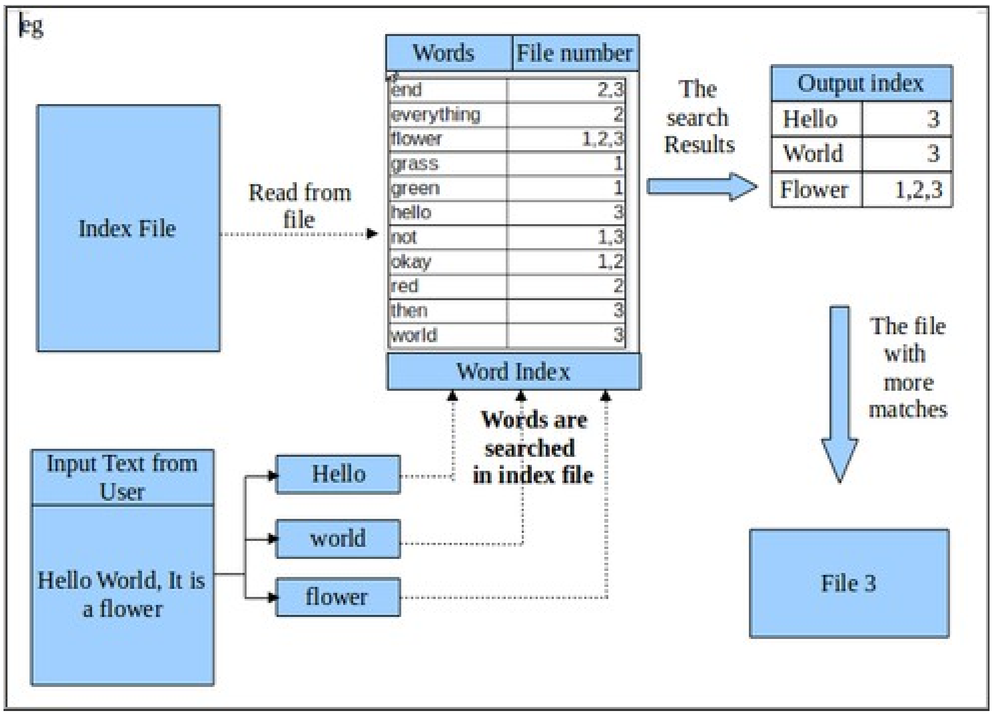
File 3: Hello world. It is a flower. then it is not the end

Words	File number
end	2,3
everything	2
flower	1,2,3
grass	1
green	1
hello	3
not	1,3
okay	1,2
red	2
then	3
world	3

Searching

Once the Indexing is over we have the Querying or Searching. The text to be searched is inputed which is parsed into words and those words are searched in the index file. To avoid the overhead of reading the file again, the file is converted back to a linkedList or hashing program, in which the words are searched. The information about the files which contain the words are collected. The ones with more matches are filtered and produced as the result.





Words	File number
end	2,3
everything	2
flower	1,2,3
grass	1
green	1
hello	3
not	1,3
okay	1,2
red	2
then	3
world	3

Output index	
Hello	3
World	3
Flower	1,2,3

Module Level

Module 1

File1- Main_Indexing

Operation : Main file

Functions : This Main program calls the the Indexing function. It reads a text file for the input files and as an output it creates a Index file in the same directory. For updating the database also this program can be called anytime.

```
int main ( int argc, char** argv );
```

File2- Indexing

Operation : Create/Update Database(Index) file

Functions :

a. Select the Files from a text file

The list of the files can be provided by storing all the file names in another file, FileList.So the names of the files which are to be documented are provided by this file. When a file is added or removed, FileList is changed accordingly. So read the file names and start indexing.

```
create_database( FILE* FileList);
```

b. Read file one by one

Open the file "FileList" and read the names of the files. This is an iterative process which reads one file name from the FileList and open that file in read mode.

```
word* read_datafile ( FILE *file, Word *WordList, char* filename)
```

Now the process of parsing starts.

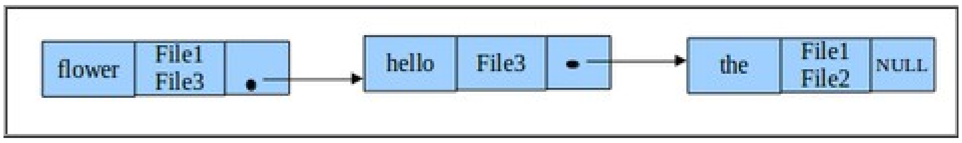
i. Select words : Character by character are read from the file. When a character other than alphabet is read it is considered as the delimiter of words. So the alphabetic characters form words. When one word is collected it now send for storing. For space reductions avoid prepositions (in, is, from, etc), conjunction (and, for etc), articles (a, an, the) etc. In text files the occurrence of these words will be more.

ii. Store in the index list along with file name in sorted order: Index List can be stored using Linked List or Hashing. Here we are implementing with LinkedList. The word is now stored in the linked list along with file names. So the Linked List is a structure storing a) word b) list of filenames c) pointer to next list. The word is inserted in the list in sorted order.

For arranging in the sorted order the words in the list are string compared with the new word to be inserted. Before inserting the word, make sure the word is not repeated. If the word is already present, select the same word node and add the file name in the file list. For the samewords in the same file no need of adding the same file name again. (instead for advanced search, you can increment the count of that word in the same file. For this the node should contain a count member which reflects the weightage of each words in each file.)

iii. Once the file reaches the end, it can be closed. And the next file file in the file list can be opened in read mode and process continues. This is done for all the files in the list. So When the files are over a very big index (linked list) is formed.

word* store_word (char* word, char* filename, Word* WordList);



c. Write the index list in the Output Index File : All the files are opened and read. Words are listed. Now this list is to be stored. For this a new file "DatabaseFile" is created and opened in write mode. Now the words along with filelist (each node contents) are written to the Database file. In the file also the word should be written in sorted order. After that the file is closed properly.

write_databasefile (Word* WordList, FILE* Databasefile);

Module 2

File1- Main-Searching

This program can be called whenever a search is needed. This program takes user input texts and produces file names in return.

File2- Searching

Operation : Search the input read from the user and produce results.

Functions :

a. Write the Index file to a index list

Before starting 'search phase', open the Database File and read the word structure and store it in a LinkedList. This helps to avoid the overhead of accessing files from file always will be reduced. So open Database file in read mode, read each word& related files info, and store in a LinkedList node. Read till the file is completely converted to LinkedList. Make sure the sorted order is stil maintained.

Word* create_indexlist (FILE* Index_file, Word* Index_List); return Index_List

b. Take input from user

Now the search phase is going to begin. To search, collect the user input text. It should be stored in a character string.

Char* input_from_user(); returns user input string.

c. Divide it into words

The character string can be tokenized into words . It can be done in the same way, the file is parsed. The non-alphabetic characters can be used as the delimiter.

Char* string_token (char* input_string); return words.

d.Search

i.Search each word in the list.

Each word is now used to search in the linkedlist. The word has to be compared with each word in the linked list. When found, the file details can be retrieved. Since the linkedlist is in sorted order, the complexity of searching the word in the complete list can be avoided. While string comparison, if the word in linkedlist is greater than the word to be search, it shows the word doesnt exists in the list,

ii. Create an output index, based on words matching.

For the matched words an output index is need to be created. For this an array or another temporary linked list can be created. In this, file name and a count is stored. For each word matched, the corresponding file's count is incremented.

```
Struct _file { char* filename; int count; struct _file next;}File;
```

```
Search (char *word, Word * IndexList, File* Output_list); returns Output_list
```

e. Select the files with maximum matching and print the results in the order.

When all the words are searched the ouput list is formed. In this the file with max count (representing maximum word match) can be selected and displayed. For advanced searched, (weightage of words stored in database linkedlist) can also be used. So the results will be more accurate.

```
display_output ( Output_list);
```

Applications

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 subfragment 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, just beyond the available RAM capacity of most personal computers today.

Coding Guidelines

- Use proper naming conventions , variables, functions, data types, file names and directories.
- Do not hard code values.
- Arrange class definition into separate headers an functions in separate .h and .c files.
- Add block comments for files, and functions. Apart from this add comments wherever applicable

References

http://en.wikipedia.org/wiki/Inverted_index

www.csee.umbc.edu/~ian/irF02/lectures/04Search-Inverted-Files.pdf

http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap03.htm