Design Document

RAJATH S 2012B5A7589P
MIHIR BISWAL 2012B2A7767P
AJINKYA KOKANDAKAR 2012B3A7463P
PRABHDEEP SINGH WALIA 2012A8PS255P

1 FEATURES

The developed search engine supports the following features:

- Ranked Retrieval
- Free-form queries
- Dynamic Indexing – Adding new documents to existing corpus
- Scalable – only limited by the size of the external memory
- HTML & TXT Documents

2 ARCHITECTURE

2.1 OVERVIEW

The Information Retrieval System is a ranked retrieval system based on the vector space model with TF-IDF weighting. The system takes a free form query string as an input and returns a list of documents in order of decreasing relevance. The entire system is developed using external memory data structures and algorithms as a result it is completely scalable limited only by the size of the external storage.

2.2 COMPONENTS

It consists of three major components:

1. HTML Parser
2. Indexer
3. Query Processor

2.2.1 HTML Parser

This is an optional albeit important component of the system.

INPUT – .HTML file(s). These input files form the corpus to be searched.

OUTPUT - .TXT file. The output file is the stripped down version of the webpage. It only contains the relevant textual information in normalized form.

WORKING – The text enclosed in the <p> tags in the HTML file is extracted. Only this text is considered relevant for the sake of simplicity. The rationale behind this assumption is that majority of the web pages store useful textual content in <p> tags.

Conversion of diacritics and accents to appropriate ASCII character set representation (e.g. Schrödinger to Schrodinger) and case folding is done during normalization.

IMPLEMENTATION – The relevant text (as defined above) is extracted using BeautifulSoup4, a python library designed for scraping web pages. The text is normalized into ASCII character set using the unidecode python library.

2.2.2 Indexer

INPUT: TXT Files. These files are a part of the corpus. They are processed further to construct the index.

OUTPUT: Inverted Index, Document Map, and Term Map – stored in the external memory as B+-trees.

WORKING – In the first step, the text is tokenized into words. The tokens are filtered to remove stop words. The remaining tokens are stemmed into terms. The frequency distribution of the terms is calculated. This is then inverted and stored as temporary files. These temporary files are merged to produce the posting list. Further, each document is allotted a document id, a unique integer to identify the document. Similarly, each term is also allotted a term id. A B+-tree of these indices is constructed. An external memory map containing the document lengths is also constructed as B+-tree. These data structures are stored on binary files for persistence.

IMPLEMENTATION – The tokenization is implemented using the 'Punkt tokenizer' in Python NLTK library along with regular expressions. Stop words are stored in a Bloom Filter for constant time checking if a given token is a stop word. If any other dictionary is used, it at least takes O (log N) for each token, where N is the number of stop words. The false positive rate of Bloom Filter is minimized to below 0.1% by allocating a large size for it. 'Snowball Stemmer' is used for stemming the tokens into terms. The files are merged using our own implementation of Logarithmic Merge. B+-trees are constructed using the STXXL library for C++. The natural language process of the indexer is achieved through a Python script which is executed by loading the Python interpreter into the C++ program through the Python C API.

2.2.3 Query Processor

INPUT: Query as a string input in the standard console.

OUTPUT: A list of documents relevant to the search query ranked in decreasing order of tf-idf score.

WORKING – The program starts by loading the four indices/maps explained in the Data Structures section. The search query is tokenized, normalized and stemmed. The term id of each term in the search query is determined by searching the term-map B+-tree. The Posting List corresponding to each of the terms in the search query is fetched. On traversing through the posting lists, the tf-idf score contribution of each term-document pair is added as a <docid, score> pair to a vector. Further, the vector is sorted based on the docid. By traversing through the sorted vector, the total tf-idf score of each document is determined and stored in a map with docid as key and score as the value. This map is then sorted on based on the score. The Path corresponding to a particular docid is obtained from the document map B+tree.

IMPLEMENTATION – The tokenization, normalization and stemming is implemented using the functions described in indexing. The Indices, Vectors and Maps are external memory data structures from the

STXXL library for C++. Sorting is carried out through the asynchronous parallel disk external sort algorithm implemented in the STXXL library.

2.3 Data Structures for Indexing

There are four major indices created from the input corpus. All the four indices are stored as B+-trees on the secondary storage using STXXL Map (B+ Tree) data structure. The indices are implemented using external memory data structures. The use of external memory data structures makes the system completely scalable. If external memory data structure are not used, the system is constrained by the amount of primary memory can store considerably less amount of data. The main bottleneck in accessing a disk is the seeking. B+ Tree is a data structure which is designed to reduce the number of disk seeks. The four indices are –

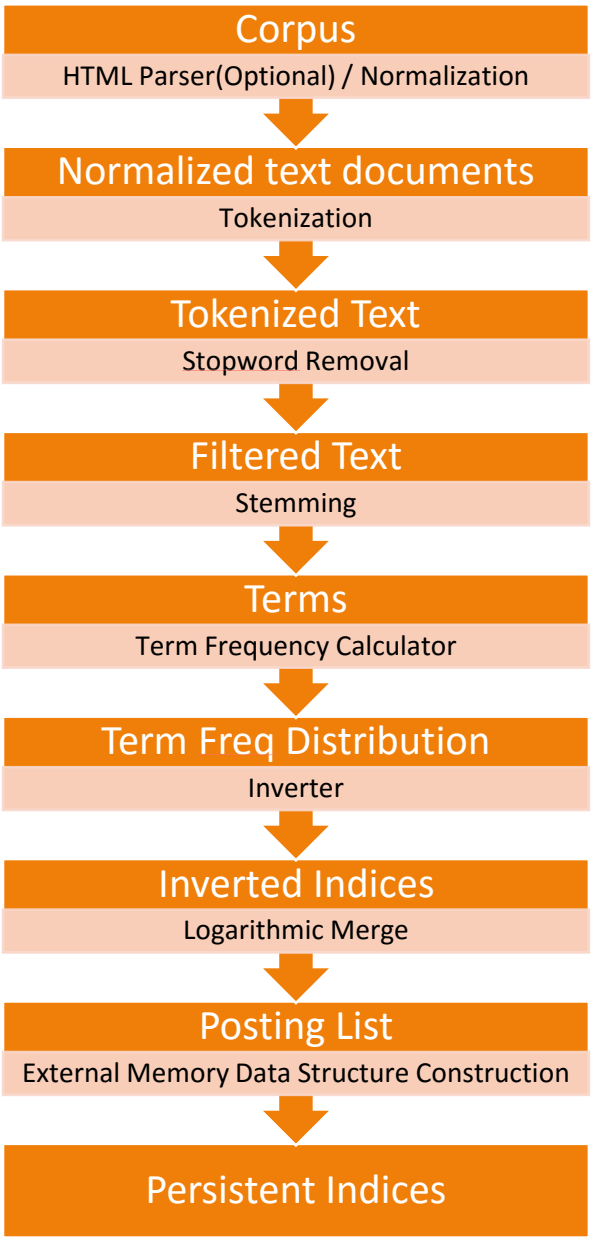
1. Term – <TermID> Map: This index stores a unique identifier <TermID> (unsigned integer) corresponding to every Term in the corpus. This is implemented as a B+-tree. Terms are padded up with spaces up to 48 characters or stripped to 48 character length before adding to the B+-tree.

2. <DocumentID> – Document Path Map: This index stores the path to a document in the corpus, identified uniquely by an unsigned integer <DocumentID>. The B+-tree for this index stores <DocumentID, SeekIndex> pairs, where the SeekIndex is a location on a binary file, at which the document path is stored.

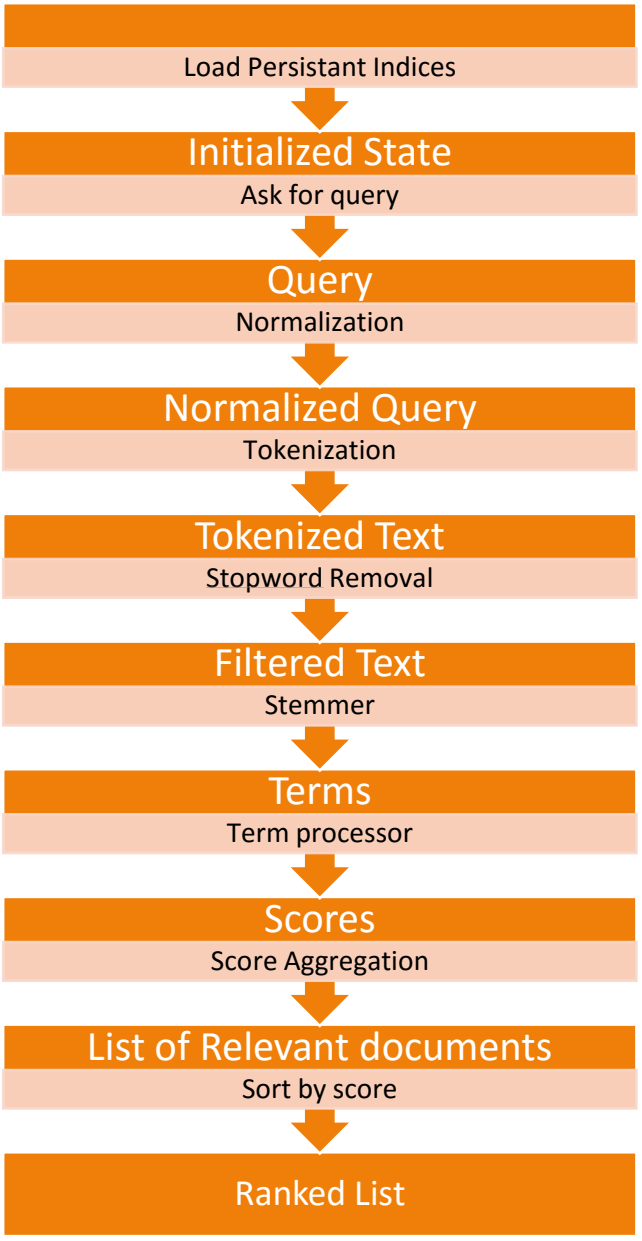
3. Inverted Index – Posting list of each term is stored in this index. Given a <TermID> as input, it stores a SeekIndex, a location on a binary file, where the posting list is stored. The Posting list consists of document frequency of the term and <DocumentID, Term Frequency> pairs.

4. Document Length Map: This is a B+ tree mapping document ID to the document length as defined in the vector space model. The map consists of pairs of <docID, docLength>. The document length is used in calculating the cosine score for a query document pair.

A Bloomfilter is used for storing stop words. This provides O(1) check if a given token is a stop word. Therefore, achieving a O(T) algorithm for filtering stop words as opposed to O(T log N) in case if any other data structure is used, where T is the number of tokens. The False Positive Rate is minimized to under 0.1% by choosing a very large sized bit vector in the implementation of BloomFilter.



Workflow: Building Index



Workflow: Search

3 VECTOR SPACE MODEL

The system is a ranked retrieval system based on the vector space model. The scoring scheme used is the tf-idf weights.

3.1 SCORE:

$$weight = tf_{i,d} \times idf_i$$

$$idf_i = \log(\frac{N}{df_i})$$

Where,

df_i is the document frequency of the ith term i.e. the number of documents which contain the ith term.

$tf_{i,d}$ is the number of occurrences of the ith term in the dth document.

3.2 LENGTH OF DOCUMENT:

$$Length = \sqrt{\sum_{i=1}^{N} \vec{V}_i^2(d)}$$

Where,

$\vec{V}_i\ (d) = tf_{i,d} \times idf_i$ is the weight for the ith term.