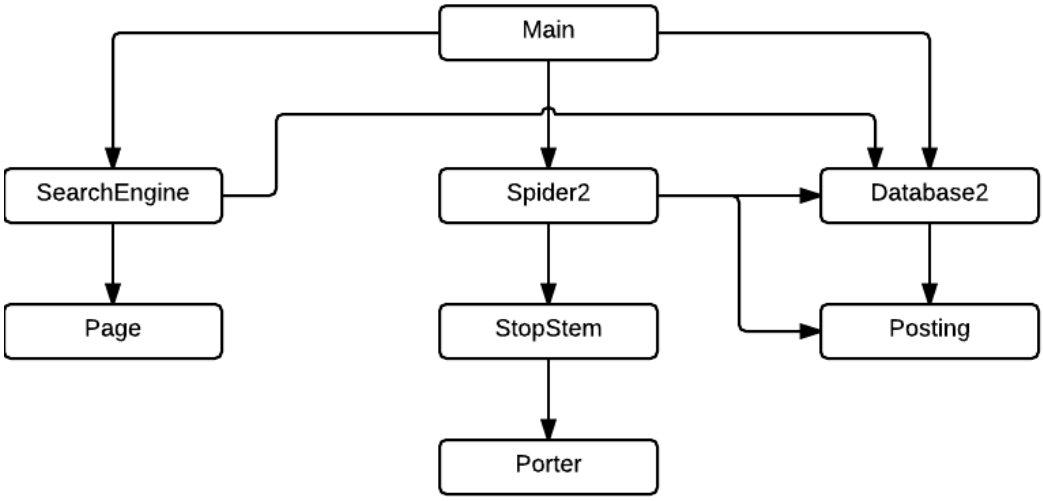
COMP4321 Search Engine Project Documentation

Alexandre Cukier | Alex Jiao | Rasmus Hvingelby

I. Overall of the system



Main

Main method of the program, gives different options to the user:

- "*Print the indexed pages to 'spider_result.txt*' ": runs the printing algorithms from the DB and creates the spider_result.txt file from the stringbuilder that the DB method getPages() returns.
- "*Start the spider and begin indexing*": call the Spider and crawl pages.
- "*Start a search and get results from the search engine*": compute a search, and print all the pages of the structure returned by the SearchEngine. Prints page information, most frequent keywords, parent page link, and children page links.
- "*Exit*": exit the program

Database

The **`Database`** class handles the connection to the JDBM database. The methods **`addPage()`** and **`addUrl()`** will respectively add a page to the pageTable and add a URL to the urlTable. The method **`getPages()`** will both return a StringBuilder with all the entries in

the pageTable and print all the entries to the console. To do this, the method uses these following helper methods to ensure the correct format of the displayed output.

- **`printTitle()`**
- **`printURL()`**
- **`printModifDateAndSize()`**
- **`printWordFreq()`**
- **`printChildLinks()`**
- **`printLineBreak()`**

The methods **`saveInvertedIndex()`** and **`saveDocumentFrequency()`** both takes a Hash Table data structure as input and then saves the structure in the invertedIndex and documentFrequency tables. Whereas the methods **`getDocumentFrequency()`** and **`getInvertedIndex()`** do the opposite operation and returns the corresponding Hash Tables.

Spider

The **`Spider`** class is used for the search engine to crawl and index the different web pages. The spider is given a start URL as parameter for the main method and recursively extracts the words and URL's from the first page. Structures are created and updated in the memory, and saved in the database when it is finished.

Crawling:

The words will be extracted in the **`extractStrings()`** method and then the words will be passed to the **`Spider`** method **`getWordFreq()`** which will remove stop words and stem the others (using the Porter algorithm) and put them in a Hash Table data structure with the words and their document frequency.

The URL's extracted via the **`extractUrls()`** will be stored in the a Linked List datastructure **`urlList`**. A while loop in the **`indexPages()`** will iterate through the Linked List of URL's until the list is either empty, or the specified maximum number of pages to index is reached, and adding the pages to the pageTable.

Indexing:

The Spider builds the system's inverted file index by extracting keywords from HTML texts. When a page is crawled from the web, we parse the document to extract out its title, body content, last modified date, children links et cetera to be stored as a page vector in the database.

For each page, the spider tests if the page has to be indexed (not yet indexed, or having been updated since last indexing). To implement the mechanism for favoring title matches, we need to have two inverted indexes to perform ranking function for both the document body and the document title. Hence, the indexer will have to perform the aforementioned operations twice

to create the two inverted indexes. It also computes the max term frequency of each of them, and creates a posting list containing the different positions of the words. The forward index is computed at the same time.

The parent/child link relations are built at the end of the indexing, taking in consideration only indexed pages. It is done by the **`saveParentChildStructure()`** method.

Search Engine

The **`Search Engine`** class performs searches. The method **`search(String rawQuery)`** is given a query, and split it into simple words and phrases (like “Hong Kong”).

It interfaces with the web interface to be used for retrieving pages from the inverted index , computing their similarity scores and ranking the pages based on the user’s query. We use the vector space model as our retrieval model. For more details about it, see the algorithms part.

Posting

The **`Posting`** objects represent the elements of the posting lists of the inverted indexes. It contains the pageId, a list of the different positions of the word, and its weight.

Page

The **`Page`** class is a representation of one page result of a search. It implements the Comparabe interface, to be ordered in the page result list. It contains all the information about the page, in particular the most frequent keywords.

StopStem

This class are being used for stemming and stopword removal. The class contains the method **`isStopWord()`** which will determine if a given word is a stop word or not, based on the stopword file which is read in the constructor. The method **`stem()`** will return a stemmed word of the given word based on Porters algorithm.

Porter

The **`Porter`** class implements the Porter algorithm used to stem words.

II. File structures (design of the JDBM database):

Schema of pageTable

This schema is used for keeping track of the indexed pages. Whenever a page has been indexed it will have an entry in this table where information about the page is kept and the words are kept as the Hash Table wordFreq with the word itself and its term frequency (wordFreq corresponds to the forward index).

PageTable is a vector that contains elements which are used for results display and computing the term weights and similarity scores.

Attributes	**Description**
pageId : int (index)	ID of the page
title : String	Title of the page
url : String	URL of the page
lastModified : String	Last modified date of the page
size : int	Size of the page (number of words)
maxTermFrequencyTitle : int	Max term frequency of the page title
wordFreqTitle : Hashtable<Integer, Posting>	A hash table with all terms in the page title and their frequencies, corresponds to forward index
maxTermFrequency : int	Max term frequency of the page body
wordFreq : Hashtable<Integer, Posting>	A hash table with all terms in the page body and their frequencies, corresponds to forward index

Schema of fowardIndex (wordFreq of the pageTable):

pageId : int (index)	posting : Posting

Mapping structures

Name	Structure	Description

URLtoPageID	url : String (index) → pageId: int	Hash table used to get the pageId of a given URL.
pageIDtoURL	pageId : int (index) → url : String	Hash table used to get the URL of a given pageId.
wordToWordID	word : String (index) → wordId : int	Hash table used to get the wordId of a given word
wordIDtoWord	wordId : int (index) → word : String	Hash table used to get the word of a given wordId

Schema of invertedIndexes

word	*postingList* : List <Posting>

Schema of the postingList

postingList is a Java object, here are its attributes:

pageId : int	ID of the page that contains the particular term
positionList : List<Integer>	A list containing the positions of the particular term. The length of the positionList is used to compute the term frequency.
termWeight : double	The weight of the particular term is computed using the formula: (tf * idf) / max{tf}

Parent/Child link relation structures

Name	Structure	Description
parentLink	pageId : int (index) → parentPageId : int	Hash table used to get the parent page ID of a given page ID
childrenLinks	pageId : int (index) → childrenPageIds : List <Integer>	Hash table used to get the children page IDs of a given page ID

III. Algorithms

Free text query algorithm

To handle free text query, it is relatively trivial as the output is the list of documents that contain any of the query terms. So we will get the list of documents for each term and take the union of them.

Phrase query algorithm

In this project, we need to handle phrase query search, i.e. *"hong kong" universities*. Hence the documents containing "hong kong" must be present in the search. This is the reason why we have word positions in the inverted index. We make use of the word positions to perform a phrase query algorithm that decides if a document contains all of the phrase query (enclosed in double quotes) in the specified order. The algorithm is as follows:

Step 1: Find all the documents that all the phrase query terms appear in, store their IDs in a list of lists
Step 2: Perform intersection of the lists of documents IDs to obtain the documents with all the phrase query terms
Step 3: We then check whether the terms are in the correct order not. For each document that contains all query terms, get the posting lists of all query terms.
Step 4: Extract out positions of each term and store them in a separate list of lists.
Step 5: Subtract i-1 from every element in the i-th list.
Step 6: We take the intersection of the list. If it's not empty, we conclude the document is matching document.

Ranking algorithm

The ranking algorithm works in line with the query algorithm. Only the collection of documents that are filtered by the query algorithms will have their ranking computed.

For our ranking algorithm, we use the popular TFxIDF algorithm which is based on the vector space model. Using the TFxIDF algorithm, the term weights of each word is computed relative to the document during the indexing phase, and are stored in the inverted index. The formula for computation is: $(\text{tf} * \text{idf}) / \max\{\text{tf}\}$.

To rank each document, we have to compute a similarity score between the document and the query. We use cosine similarity measure in the vector space model. Generally speaking, the similarity score of a document is the sum of the weights of the query terms that appear in the document, normalized by the product of Euclidean vector length of the document and the query length. The ranking algorithm is as follows:

```
Initialise scores[], magnitudes[]
For each term Q_i in Q do
      Look up Q_i from inverted index
      If not found then
            continue
      end
      If found: retrieve postings list for Q_i
            For each document D_j on the postings list do
                  // Compute partial score between D_j and Q
                  score [D_j]+= partial score [ D_j]
                  Update document magnitude
            end
      end
end
Perform normalization
```

Mechanism for favoring title matches

Because the importance of terms in title and body are different. If a page's title contains the query term, then the page should have larger chance to be returned to user compared with a page which only contains the query in the page body. In order for this, we should treat title and page body differently.

In implementing the mechanism for favoring title matches, the ranking score of a page should be based on page title's similarity (denoted as s_t) and body's similarity (denoted as s_b) with the query term, then the total similarity score for each document is w1*s_t+ w2*s_b, where w1 and w2 are parameters and w1 should be larger than w2.

Based on our experience in testing our search engine, we chose w1 to be **0.2** and w2 to be **200** as our body score weight and title score weight respectively as they give the most optimal set of search results.

w1 = 0.5 , w2 = 100

```
Free text query: love


Page 22: Fighter (2001), score: 19.772662137069318
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/10.html
Wed, 27 Oct 2010 02:33:18 GMT, 13199
neighborkyunggeun 218; jongil 74; spenc 56; titl 53; snuffl 51;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
------------ Children links ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
--------------------------------------------------------------------


Page 51: Love Reinvented (2000), score: 17.176638224610798
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/39.html
Wed, 27 Oct 2010 02:33:18 GMT, 6494
jongil 40; segment 15; imdb 11; bridgetthi 10; user 10;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
------------ Children links ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
--------------------------------------------------------------------


Page 100: Record of Lodoss War: Chronicles of the Heroic Knight (1998), score: 13.967843894993925
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/88.html
Wed, 27 Oct 2010 02:33:18 GMT, 9657
jongil 77; voic 53; english 39; independient 16; episod 15;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
------------ Children links ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
```

w1 = 0.2, w2 = 200
(top 3 pages are all relevant)

```
Free text query: love


Page 51: Love Reinvented (2000), score: 28.5484582071462
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/39.html
Wed, 27 Oct 2010 02:33:18 GMT, 6494
jongil 40; segment 15; imdb 11; bridgetthi 10; user 10;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
------------ Children links ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
--------------------------------------------------------------------


Page 57: The Love Letter (1999), score: 27.169813874653293
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/45.html
Wed, 27 Oct 2010 02:33:20 GMT, 3511
setup 26; lazi 23; jongil 19; snuffl 15; hittin 14;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
------------ Children links ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
--------------------------------------------------------------------


Page 88: I Love Lucy: Season 2 (1952), score: 27.169813874653293
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/76.html
Wed, 27 Oct 2010 02:33:20 GMT, 3530
setup 27; jongil 18; titl 15; peebl 15; hittin 14;
------------ Parent link ------------
http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/Movie/293.html
```

Screenshots of favor mechanism with different title and body scores (in Eclipse console)

IV. Installation procedure:

See the readme.txt file.

Conclusion

The project should be started earlier, so that there would be more time to debug and optimise the search results. One weakness of our search engine is that the data structures and algorithms are not optimised for big data sets. So the query time will increase if the number of pages indexed grows exponentially. The strength of our engine is that our phrase query search results are very relevant. We tested our search engine using various esoteric terms and we are pleasantly surprised at the relevance of the search results.

In the future, we hope to implement PageRank algorithm to make our search results even more relevant. One thing we hope to do differently when we first started is that we need to plan out the data structures of the file structure more carefully, because we hit multiple road blocks due to inefficient storage of data.