**Overall design of the system**

This system consists of multiple parts. First, we define a data manager object (DataManager. java) for the databases. It contains the record manager and the hash table. We also have spider (Spider.java) to fetch all the pages with a given URL using the breadth-first strategy. We get all the child links, parent links, the URL of each page into an index database. HTML parser libraries are used in fetching the pages, extracting the keyword and the links of the web pages. Then we use an indexer to extract keywords from each page. We removed all the stop words from the pages and turn the words into stems using Porter’s algorithm before storing them in another database. We also create a page object (Page.java) for each page which includes functions to get the page information.

The main part of this system is the search engine. We get the query typed by the user and then stem them. Both single words and phrases (Phrase.java), which should be specified with double quotes, are accepted. After that, we can compare the stemmed query with the stemmed words in the database to check if there is a match in any documents. We match the query and the keywords in documentation by calculating their scores. A match in the words in the title will give a higher score. The overall score is the score from the body part and the score from the title part multiplied by 5. We formulate the score in this case also because we do not want the outcome scores to be too small. Documents with a higher score will be shown first on the user interface. We use JSP page to pass the query string to the search engine. To show the result of the whole system, we designed a nice web interface to accept the query from users using a text box and submit the query to the search engine. The result will be shown in descending order of the related pages’ score. In addition to the page information and the URL, the top 5 stemmed words with the highest frequency will also be shown.

For this system, we also have a test program (TestProgram.java) for testing. All the new implementations will be tested first before we applied them in the search engine system.

**The file structures used in the index database**

We use lots of index databases in this system to store the information. Databases are mainly used in two areas. One of them is related to the page information and the other one is related to the keywords in pages. First, we have several databases related to the pages.

**pageID(URL, pageId)**: When we first retrieve the pages, we store the URL that we get in the database. The purpose is to assign each page a page ID and get the page ID when we only have the URL, such as checking whether a page have been retrieved or not.

**pageURL(pageId, URL)**: This database is the reverse version of pageID. We can use this database to get back the URL by using the pageId. One of the usage examples is to find out the child page URL using this database. When we need to find out the child link of a page, what we get from another jdbm is a list of page ID of child pages. To print out the child link URL on the interface, we need to convert it to URL first.

**pageInfo (pageId, page)**: This one store the page information, such as the page size, page title and the page link, of the indexed pages with the unique page ID. It is useful when we need to print the page information on the result page, such as the last modification time, parent and child links, etc.

**childLinks(pageId, Vector<ChildLinks>)**: This database stores a list of child page ID for each page. The key of this database is the page ID. We need to use the pageID to find out the pageId of a specific page so to use the page ID in the childLinks database. We store the child links to a list when we retrieve those pages in Spider.

**parentLinks(pageId, Vector<ParentLinks>)**: We store a list of parent links of a page with the page ID as the key. It is quite similar to the childLinks.

Other databases are related to the words that we get from the web pages.

**wordID(word, wordId)**: This database stores the keyword that we get from the page. Whenever we get the keywords from a page, we will store those words to this database and assign a word ID for each. When we get the query from the users, we need to chop the whole query into words and check if the words exist in this database. Then we can use the word ID to find the pages containing this word with help from other databases.

**Word (wordId, word):** This database stores all the word ID with the corresponding word.

**bodyWord(wordId, Vector<{pageId,tf}>)**: This database stores the word ID for the words that are found in the body part of the page. This will be used after we get the word ID of a word from the query. We can check which pages contain that word in the body part.

**titleWord(wordId, Vector<{pageId,tf}>)**: This one looks similar to bodyWord, but this database is stored the word ID of the keyword in the title. It uses the word ID as the key and stores the vector which has the page ID and term frequency. This will be used after we get the word ID of a word from the query. We can check which pages contain that word in the title.

**pageBodyMaxTF(pageId, max. body tf)**: This database stores the maximum term frequency in the body part of each page. The key is the page ID. We use this database when we need to calculate the term weight of each word in the body since the term weight formula is .

**pageTitleMaxTF(pageId,max. title tf):** This database is quite similar to pageBodyMaxTF. It stores the maximum term frequency in the title of each page. It is also used for calculating the term weight of each word in the title.

**pageBodyWord(pageId, Vector<keywords in body>)**: This is a forward index database. This database stores the keywords in a list after we retrieve and stem the words in the page body. It is used in the Query.java to get the list of stemmed words of a page.

**pageTitleWord(pageId, Vector<keywords in title>)**: We only create this database for the test program. It is not actually used in the system.

**Testing of the functions implemented; include screenshots if applicable in the report (2 pages)**

**Conclusion: What are the strengths and weaknesses of your systems; what you would have done differently if you could re-implement the whole system; what would be the interesting features to add to your system, etc.**

Our systems have a good search speed but with the cost of relatively slow indexing speed. For each page, we store many information related to the page, such as its parent links, child links, stemmed keywords together with their occurrence frequencies (i.e. the forward index). Also, for each stemmed words, we store exactly where the word appears in the pages, i.e. the word positions, to support phrase detection. The posting list is long but it is fast in terms of retrieval. Using the advantage of inverted index, getting an entry from a long list is exceedingly fast. Therefore, the process of calculating the term weight and also the phrase weight (if any) is also fast. The score object we defined is comparable with itself so that the sorting is very easy, simply use the sort function from Collections class.

On one hand, due to the time limit and the tight schedule during this semester, our group cannot implement more advanced features that we want to implement originally. Some interesting features would be: suggesting similar pages in terms of related topics; wild cards search (e.g. when user only knows a part of the keyword instead of the whole word, the user can search like “comput\*”); categorize the indexed pages and build a directory system for users to search for a specific topic, etc.

On the other hand, we have thought for a moment to use a more modern web programming languages, i.e. Node.js to implement the whole system. However, without prior experiences on large-scale server-side JavaScript programming and we have other courses and projects to handle, we gave up this idea and go for implementing the search engine using Java in which we are given more instructions to work with.

**Workload Distribution in Final Submission (Group #7):**

Chan Man Yee (20278380): JSP, Whole Interface, Sorting of top 5 keywords, documentation(Testing)

Lui Ka Kit (20270833): Spider, Indexer, Query, Phrases, Sorting , Score, Posting, Forward Posting, Documentation (Strength & weakness)

Tsang Cheuk Ling (20277506): Term Weight, Query (basic version), Test Program, retrieve and store parent links into database, Documentation (overall design and databases)