Query Design Spec

In the following Design Module we describe the input, data flow, and output specification for the query module. The pseudo code for the query module is also given.

The following are included in this Design Spec:

(1) Input: Any inputs to the module
(2) Output: Any outputs of the module
(3) Data Flow: Any data flow through the module
(4) Data Structures: Major data structures used by the module
(5) Pseudo Code: Pseudo code description of the module.

(1) *Input*

Command Input

./query [INDEX_FILE] [HTML_DIRECTORY]

Example command input

query ../../indexer/index.dat ../../crawler/data

[INDEX_FILE] ../../indexer/index.dat
Requirement: The file must exist, and contain the InvertedIndex built from indexer.
Usage: The query needs to inform the user if the file cannot be found.

[HTML_DIRECTORY] ../../crawler/data
Requirement: The directory must exist.
Usage: The query needs to inform the user if the file cannot be found.

(2) *Output*

From [INDEX_FILE], the query will build an InvertedIndex, take in user query input, use the InvertedIndex to find matching documents, and output the url and doc_id to stdout.

(3) *Data Flow*

[INDEX_FILE] is loaded to create an InvertedIndex of word to document pairs. Then, it waits for the user to input a query. It takes this input and reads by word. If the word is not a logical operator (AND, OR), then it searches the InvertedIndex for all DocumentNodes that contain the word and store them into temp_list. If the word read is AND, it reads in the next word and does an intersection operation of matching DocumentNodes. The result is stored back into temp_list. Successive words with no operator are construed to be an intersection operation. If the word is OR, then temp_list is flushed out to final_list through a union operation.

The above process is repeated until all the words in the query have been parsed through. Then, the query flushes out temp_list to final_list one last time. Next, the query takes the final_list containing matching DocumentNodes and sorts it by rank, from highest frequency to lowest frequency of occurrence of query.

Afterwards, the query reads in the URL address of each document in final_list from [HTML_DIRECTORY]. This information, together with the document ID, is outputted to stdout.

(4) *Data Structures*

WordNode – holds information about a word, such as in which document it appears, and how frequently.

DocumentNode – holds information about a document that a word appears in.

InvertedIndex – holds a table of all the words, where each word points to a list of documents that it appears in. This is essentially a HashTable.

(5) *Query* Pseudocode

```
// check command line arguments
    // Inform the user if arguments are not present or invalid

// initialize data structures / variables

// read in [INDEX_FILE] and create an InvertedIndex.
```

```
    // Wait for query input from stdin.
        // check query line for invalid input.
        // get list of documents containing the query.

        // sort the list of documents by frequency, from highest to lowest.
        // display the doc_id and url of the sorted list of documents.

        // free resources

    // free resources
```

Query Functionality

Below are descriptions of functions used to build the Query. Functions already provided, or built in Crawler/Indexer are not shown.

```
/*
 * GetLinks - get all matched DocumentNodes for a given query.
 * @line: query line to be searched for.
 * @Index: InvertedIndex containing all word-document pairs.
 *
 * Returns 1 if successful.
 * Returns 0 if not successful.
 *
 * Pseudocode:
 *     1. Read in each word of the query.
 *     2. Save all matched DocumentNodes in temp_list.
 *     2. If an AND is passed, or there is no logical operator between two words,
intersection operation is done on temp_list.
 *     3. If an OR is passed, data in temp_list is flushed out to final_list using the union
operation.
 *     4. Continue until end of line.
 */
int GetLinks(char *, HashTable *)
```

```
/*
 * And - Perform intersection operation.
 * @word: word to get new list of matching DocumentNodes from.
 * @Index: InvertedIndex containing all word-document pairs.
 *
 * Pseudocode:
 *     1. Find the start of the list of matching DocumentNodes for the word.
 *     2. For each DocumentNode in temp_list, go through the new list looking for doc_id
matches.
 *     2. If there is a match, keep that DocumentNode in temp_list and increment
frequency.
 *     3. If there is no match, delete that DocumentNode from temp_list.
 *     4. Continue until end of temp_list.
 */
void And(char *, HashTable *)


/*
 * Or - Perform union operation.
 *
 * Returns 0 and terminates.
 *
 * Pseudocode:
 *     1. If final_list is NULL, set final_list to temp_list, temp_list to NULL, and return.
 *     2. For each DocumentNode in temp_list, go through final_list looking for doc_id
matches.
 *     2. If there is a match, keep that DocumentNode in final_list and increment
frequency.
 *     3. If there is no match, create a copy DocumentNode and add to end of final_list.
 *     4. Continue until end of temp_list.
 */
int Or()


/*
 * Sort - Sort the list of DocumentNodes by rank, from highest to lowest.
 *
 * Pseudocode:
 *     1. Put DocumentNodes into an array.
 *     2. Sort the array by rank.
 *     2. Refactor sorted array into list of DocumentNodes.
 */
void Sort()
```

```
/*
 * key_compare - key compare function for qsort.
 * @e1: element to compare.
 * @e2: element to compare.
 *
 * Returns -1 if e1>e2, 1 if e1<e2, 0 if e1==e2.
 *
 * Pseudocode:
 *     1. Get the frequency of the two passed DocumentNodes.
 *     2. Return appropriate comparison value.
 */
int key_compare(const void *, const void *)


/*
 * FreeList - free memory of DocumentNode lists.
 * @choice: the type of list to free. If 0, free temp_list, and if 1, free final_list.
 *
 * Returns 0 and terminates.
 *
 * Pseudocode:
 *     1. For the appropriate list, loop through each DocumentNode.
 *     2. Free DocumentNode and move on to the next one.
 *     3. Free until list is empty.
 */
int FreeList(int)


/*
 * Display - display query matches to output.
 *
 * Returns 1 if successful.
 * Returns 0 if not.
 *
 * Pseudocode:
 *     1. Get each DocumentNode of final_list.
 *     2. Get filename and open a stream to that file.
 *     3. Read in the URL address from stream.
 *     4. Output doc_id and url to stdout.
 *     5. Cleanup memory and close stream.
 */
int Display()
```

Query Error Conditions / Boundary Cases
---

1. This program checks that there are 2 arguments passed.
2. This program checks that the [INDEX_FILE] exists.
3. This program checks that the [HTML_DIRECTORY] exists.
4. This program checks that the query line is not empty.
5. This program checks that the query line only contains ASCII characters separated by one whitespace.
6. This program checks for invalid input cases regarding logical operators
    - successive logical operators
    - operator in the beginning of the query line
    - operator at the end of the query line

Query Test Cases

Please refer to the log file produced from QEBATS.sh and queryengine_test.c.