Software Development for Information Systems
Inverted Search Engine

Georgios Nikolaou
sdi1800134

Nefeli Tavoulari
sdi1800190

Fall Semester 2021

HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

Contents

1 Introduction

The current project is a modified version of The SIGMOD 2013 Programming Contest. The application tackles the task of receiving a stream of documents and queries and matching them up, utilizing different metrics. The goal is to minimize the system response time.

The program is implemented in C++ language, using Object-Oriented Programming, taking advantage of Encapsulation, Data Security and Organization.

2 Methodology

2.1 Version Control

The code we implemented can be found in Github.
We used Version Control to keep track of our changes, make sure that our code is safe and most of all, to be able to collaborate as a team. In our repository we have included a Makefile, to automate the build and testing stage of the program.
Communicate with the application using the following CLI commands:

- **make** (compile)
- **make run** (run)
- **make valgrind** (check memory)
- **make clean** (clean)
- **make test** (compile tests)
- **make run_test** (run tests)
- **make valgrind_test** (check memory in testing code)

Furthermore, the repository includes a README file, where we explain basic concepts of the project.

2.2 Testing

Also, we have performed Unit Testing, using Acutest, which helped us realize the importance of test-driven programming in creating efficient, maintainable and quality software. In our tests we tried to brainstorm of all possible extreme cases to make sure that our program does not crash, but deals with exceptions.
In addition to that, we experimented with Github Actions, creating a Continuous Integration pipeline, to automate the workflow and ensure there are no bugs in the new features we add each time.

Finally, we used Valgrind to find memory leaks and thus, ensure that we are building a stable and quality software.

3 Implementation Description

The classes we designed:

- **Query** It contains a one-dimensional array of characters, where the words from the input are stored. We chose to use a 1d array, instead of a 2d, in order to save time by minimizing the times we read from disk, since the whole array can be stored in RAM (contiguous allocation). So, to access a word saved in Query (getWord function) instead of array[i][j] (2d array hypothesis) we return $arrary[i*(MAX\_WORD\_LENGTH+1)]$ .The class is composed of a constructor, destructor some getters and also some print functions for debugging. Also, it is made with respect to the limitations given in the core.h file. We have made the assumption that the input has a specific form, words that are splitted by spaces (one or more). The words can contain any type of character and in case the word is too big or too small or in case too many words are given, the extra/inappropriate words are ignored, causing no issue to the execution. In order to copy blocks of memory from the input to the Query, we used memcpy, which is the most efficient way to perform such a task. Also, we used strtok to handle word extraction from strings. Exception is thrown if the input is NULL.

- **Document** It also contains a one-dimensional array of characters. The same as for the Query class applies here.

- **Entry** It contains a word and a payload (for now, it is a pointer to void) and a pointer to the next entry. The class is composed of a constructor, destructor, some getters, setters and also some print functions for debugging. Exception is thrown if the input is NULL.
- **Entrylist** It contains a pointer to an entry, which is the head of the list of entries. When adding an entry to the entrylist, the new entry always becomes the head of the list, pointing either to NULL (if it is the first item of the list) or to the old head. The class is composed of a constructor, destructor, some getters, setters and also some print functions for debugging. Exception is thrown if the input is NULL. When deleting the entrylist, we ensure that all the entries are deleted too.
- **HashTable** We chose to use a HashTable for handling the deduplication task of the Document. After some research, we realized that this is one of the most efficient ways to reach our goal. Deduplication using Hash Table has a complexity of O(n), while the same task using e.g. Merge Sort has a complexity of O(nlogn). The HashTable receives words and stores them in the appropriate bucket, after passing them through a simple hash function. Of course, because of the fact that we are dealing with strings, we expect to end up with buckets that contain different words with the same hash. So, other than this process, we also use Binary Search each time we insert a word into a bucket, to keep the bucket sorted, so that it is easier to figure out if the word already exists in the bucket. In this case, we do not add it again. The buckets are represented as a 2d array of words, where each row refers to each bucket and each column refers to each word of each bucket. We have defined a macro for the number of buckets, which we can of course change in the future. What we wanted to achieve by this number is that the words per bucket are less than 80% of the number of buckets, which as we learned from our research, is a good rule of thumb. As far as the number of words per bucket is concerned, we have also set a macro, initialized to 100 and every time more space is needed in a bucket, we use reallocation. The class is composed of a constructor, destructor, some getters, setters and also some

print functions for debugging. Exception is thrown if the input in any of the functions is NULL.

- **Queue**
- **childQueueNode**
- **TreeNodeList**
- **IndexNode**

Some functions we used:

- **ExactMatch**
- **HammingDistance**
- **EditDistance**
- **Deduplication** It returns a pointer to a hash table which stores the unique words of the Document. It is responsible to call the hash function for each word of the Document and then add it to the Table.
- **Interface** The interface functions were built as requested, which call the corresponding class functions.

4 Design Assumptions

5 Remarks

We would like to note that some parts of the Hash Table could not be tested using Unit Testing because, obviously we can not figure out two different words with the same hash. So, we can not use Unit Testing for a bucket of more than one words. However, of course, we have tested that everything functions well and with no memory leaks whatsoever, by adding the same word in the same bucket more than once.In this way, we were able to make sure that the reallocation works well too.

6 Challenges

7 Conclusion