Software Development for Information Systems
Inverted Search Engine

Georgios Nikolaou
sdi1800134

Nefeli Tavoulari
sdi1800190

Fall Semester 2021

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, Abstraction, 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

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 software.

2.3 Directory Structure - Code files only

```
repository
├── include
│   ├── core.h
│   ├── index.hpp
│   ├── structs.hpp
│   └── utilities.hpp
├── src
│   ├── main.cpp
│   ├── index.cpp
│   ├── structs.cpp
│   ├── utilities.cpp
│   └── core.cpp
└── tests
    ├── tests.cpp
    └── tests.hpp
```

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. 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. The string input of the Query "consists of a space separated set of words. The length of any word will be at least MIN_WORD_LENGTH characters and at most MAX_WORD_LENGTH characters. The number of words in a query will not exceed MAX_QUERY_WORDS words. "query_str" contains at least one non-space character. "query_str" contains only lower case letters from 'a' to 'z' and space characters", as stated by Sigmod. The Query struct also contains a match_type (Exact, Hamming, Edit Distance) and a match_distance (0,1,2,3), which indicate the kind of match we should apply. Finally, this struct has also a boolean array with the size equal to the number of entries of the query, that indicates which entries have matched with the words of a document. This array is set to false by default and is set again to false when we try to match it with another document.

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

- **Entry** It contains a word, a payload and a pointer to the next entry. The payload is a list that stores the query ids and thresholds that correspond to the queries that contain the word of the entry.The class is composed of a constructor, destructor, some getters, setters and also some print functions for debugging. Moreover, there are some methods that are related to the pay-

load functionality, such as deleting a queryid from payload or adding a queryid to payload. 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.

- **DocTable** We chose to use a DocTable 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 DocTable 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.

- **QueryTable** This struct is a hash table where all the queries

are stored. When we want to retrieve aquery in order to delete it (EndQuery), we can find it easily through the hash table,according to the Query Id.

- **EntryTable** Just another hash table, where the entries are stored, according to their words, to perform exact match. Whenever a query is deleted (EndQuery), we delete its queryid from all the entries' payload, with not much cost. This struct also provides a method, wordLookup, which returns an entry list of all the entries which matched with words of a specific document.

- **ResultTable** Another hash table which stores the query ids corresponding to the entries returned by the lookup function. We used a hash table for this operation in order to deduplicate the results and end up with a list of the query ids which matched with a document. That means that all of the query's entries matched with words from the specific document.

- **Stack** This class acts as an interface with stackNode, to simplify the process of storing and retrieving the children of an IndexNode that fit the pruning criteria ([d-n, d+n]), when searching through a tree with lookup entry index. It contains a pointer to the first of the stackNodes, and features a constructor, methods to add and remove items and method to return its size. When adding a new indexNode to the Stack, it is placed at the head of the list and also removed from the head when it is time to retrieve a node. We chose this type of data structure to minimise the time it would take to retrieve an item from the structure, as using ,for example, a linked list, would require us to traverse the whole list which we expect to have significant size, and of course using static structures is out of the question, as we do not know the size of the input.

- **StackNode** Each object of this class contains a pointer to an indexNode object and a pointer to the next stackNode (if it exists). It is managed by a Stack object, and for this purpose features a constructor, a pop method that deletes the object and returns its contents and its next node and a getNext method, to return its next node.

- **IndexList** This class is used as part of the IndexNode, in order to store each node's children nodes. It contains a pointer to the next indexList node, a pointer to the indexNode it is storing and the distance of this indexNode from its parent. The objects of this list are sorted with regard to distance from their parent node, from lowest to highest, each one being created directly where it should belong, after the node with the next smallest difference and before the one with the next biggest. This class features a constructor, getters for the various contents, a print method for debugging and the addToList method, that manages the creation of new nodes in the correct place in the list. The reason for selecting this data structure is twofold. The first reason is minimising memory use; While using a static array of pointers was an option, as the maximum number of children is implicitly known (we are given the maximum size of a word as 31, so the maximum number of differences two words can have is at maximum 31), doing this would require that every node takes up the maximum amount of space. The second reason in speed on lookup. As the list is sorted with regard to distance, and the criteria for pruning sub-trees when searching through the tree are based on distance, we can stop examining the list of children when the first one does not meet the criteria, being sure that the next one certainly will not fit them, as it is more distant to the parent than the one we just turned down.

- **IndexNode** This class is the main building block of the tree. It stores the enumeration of the distance function of the tree, to be used when creating new nodes or searching in order to calculate the distance between nodes, a pointer to the entry it contains, and a pointer to a list of its children if they exist. It features a constructor and destructor,getters for its members and a print function for debugging purposes. It also features addEntry, a function that manages the insertion of new entries, which is called in a recursive manner , by calling the addToList method described above, which in turn calls add addEntry on the appropriate child with the same difference to the parent node as the given word (if it exists). This way, the given entry is sent down the appropriate sub-tree until it can be assigned as a child node to an existing node.

The main functions we implemented:

- **ExactMatch** This function simply returns True when the two given words are exactly the same, false otherwise.
- **HammingDistance** It calculates the Hamming Distance of two given words. Generally we can only define the Hamming distance of two words of the same size, so in order to accommodate words of different sizes, we made the assumption that the distance of two words of different size can be calculated as follows: For the common length of the words calculate the Hamming distance normally, and count every extra character of the largest word as one difference.
- **EditDistance** It calculates the Edit distance (Levenshtein distance) between two given words, using the Wagner–Fischer algorithm.
- **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 corresponding bucket of the table.
- **Interface** The interface functions were built as requested, which call the corresponding class methods.
  - **InitializeIndex** Initilization of some global class instances, such as the indices, one for the exact match(entry hash table), one for the edit distance, and 27 for the hamming distance(for every word size). Also, the query table, where the queries are stored and an entrylist which stores all the entries, so that we can find and delete them in the end.
  - **StartQuery** This function creates a query and adds it to the global Query Hash Table. Furthermore, it adds all its entries to the corresponding index, according to the size of the words and their match type. For this, it calls the addToIndex function.
  - **EndQuery** This function deletes a query from the Query Hash Table, tracking it down easily, using its id, and also makes sure to delete this id from the entries' payload, from all the indices, calling the removeFromIndex function for

every query word. If the payload becomes empty, we do not delete the entry from the index, because that would cause some trouble (eg tree children). Instead, we consider it as deactivated and we move on.

- **DestroyIndex**
- **MatchDocument** It creates a new document, it deduplicates it and it searches if the document words match with any query words. It accomplishes that accessing all indices and getting an entrylist which contains all query entries that matched with at least one document word. Afterwards, using the Result Hash Table, and by checking the matchedEntries boolean array in the Query struct, we are able to find the queries, of which all the entries matched with some document words. If only some query entries matched, then this is not a match and the boolean array of this specific query should be set back to false, so that it is ready for the next search with a different document. Finally, the results are added to a list, which is then processed by GetNextAvailRes.
- **GetNextAvailRes**

4 Final 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.

We would also like to note that the input of MT_EXACT_MATCH is considered invalid when building an index with build entry index, because even though it belongs to the enumeration "MatchType" defined in core.h it is not possible to use it to create a B-K Tree.

Furthermore, with regard to the destruction of the entries contained by an entry list and the index created by using it, we consider it as the responsibility of the entry list, so the entries are destroyed when the entry list is destroyed, the destruction of the tree has no effect on the entries.