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 and techniques.

The metrics:

- Exact Match
- Hamming Distance
- Edit Distance

The techniques:

- Hashing of the entry words for exact matching
- BK Trees for edit and hamming distance

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.
Communication with the application is accomplished using the following CLI commands:

- **make** (compile)
- **make run** (run)
- **make valgrind** (check memory)
- **make helgrind** (detect data races)
- **make drd** (detect data races)
- **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 basic concepts of the project are explained.

2.2 Compilation

The flags used in the compilation of the program:

- **-g** : It tells the compiler to generate and embed debug information.
- **-pthread** : It enables the use of the pthread library.
- **-Wall** : It enables all compiler's warning messages.
- **-O3** : It offers optimization for code size and execution time.

- **-fPIC** : It ensures that the code built into shared libraries is position-independent, so that the shared library can readily be loaded at any address in memory.
- **-std=c++11** : It enables c++11.
- **-I** : It adds include directory of header files.

2.3 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.

2.4 Directory Structure - Code files only

```
repository
├── src
│   ├── main.cpp
│   ├── index.cpp
│   ├── structs.cpp
│   ├── utilities.cpp
│   ├── core.cpp
│   └── jobscheduler.cpp
└── tests
    └── tests.cpp
```

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_r to handle word extraction from strings, in a thread safe way. 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. It consists of an id and a 1d array of characters.

- **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 payload functionality, such as deleting a queryid from payload or adding a queryid to payload. Exception is thrown if the input is NULL.

- **payloadNode** It contains the QueryID of a specific query, the distance within which a word is considered a match for this query, and a pointer to the next payloadNode, if it exists. Since the list of payloadNodes is created in an descending order by the appropriate functions of the corresponding Entry, time can be saved during lookup were the search starts from the most lax distance requirements and stops on the first payload node whose requirements cannot be met, being sure that the following nodes' requirements will be as strict as, or stricter than, the requirements of the last node.

- **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 macros for

the number of buckets and for the number of words per bucket. What we wanted to achieve by this number is to minimize the number of words per bucket and the number of collisions . The words per bucket are initialized to 10 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 methods is NULL.

- **QueryTable** This struct is a hash table where all the queries are stored. When we want to retrieve a query in order to delete it (EndQuery),or get some property, 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.

- **matchedQuery** Each object of this class stores the QueryID of a Query that matched a word with an entry in the index. This allows for the easier access to the queries that have any possibility of being matches to the document, as well as for resetting the status of matched words for any query that had at least one matched word. The matchedQueryList class serves as an interface for the management of the list.

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** Deallocation of the aforementioned structs.
- **MatchDocument** It creates a new document, it deduplicates its words and it searches if they 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. In any case, the boolean array of these specific queries should be set back to false, so that they are ready for the next search with a different document. Finally, the results are added to a list, which is then processed by GetNextAvailRes.
- **GetNextAvailRes** This function returns the results of the matching, that means the query ids of the queries that matched with a document, the document id and the number of results. The query ids were sorted using Merge Sort.

4 Design Choices

Some design choices we made in order to improve the time results are the following:

- Allocating continuous blocks of memory, using 1d arrays, instead of 2d, for example to store the query and the document strings. In this way the arrays are kept in cache.
- Using memcpy and strcpy to copy strings/bytes and avoiding redundant iterations. Also, statically allocating data structures, when the size was predefined.
- Using hash tables for deduplication (e.g. DocTable, ResultTable) or to achieve faster lookup (e.g. QueryTable, EntryTable). Lookup, insertion and deletion in a hash table without collisions has a complexity of O(1), which is very fast, in comparison with other methods.
- Keeping the payload nodes sorted in descending order based on the matching distance, so that during lookup, we can stop the process, the moment the first payload node does not meet the threshold.
- Using Merge Sort to sort the results for GetNextAvailRes, which is one of the most efficient algorithms for such a task.
- In order to obtain correct results, each thread needs to have exclusive access to the Query Table, because this is where we store the information of whether an entry has matched. That is why we need a local query table, which is cloned using the cloneQueryTable function. After every batch of query insertions and deletions, the global query table is cloned once for every thread.

5 Parallelization

By changing the value of a macro in core.h we, can change the multithreading mode, that means the parts of the program to be parallelized. The parallelized parts of the execution are the following:

- MatchDocument : Parallelization of the entry lookup using a thread for each document.
- StartQuery : Parallelization of the entry insertion to the 29 indices.
- EndQuery : Parallelization of the entry deletion in the 29 indices.

For this process, we created the following classes and functions:

- JobScheduler, Job, JobNode, JobQueue : A job is composed of a pointer to a function and the corresponding arguments. A jobNode is a job and a pointer to the next job inside the JobQueue. The JobQueue is a FIFO list and the JobScheduler is a class consisting of the JobQueue and the threads.
- MatchDocumentArgs, StartQueryArgs, EndQueryArgs : the arguments for each job. In this way, we can pass arguments as a pointer to void into the job constructor.
- MatchDocumentJob, StartQueryJob, EndQueryJob : the jobs that are parallelized.
- execute_all_jobs : the main thread functionality. It executes jobs, popping them from the JobQueue, until the JobQueue is empty.

Also, we used mutexes, condition variables and also some global variables, that helped us synchronize the threads.

- queueLock mutex : protecting the scheduler's jobQueue.
- previousOperationLock mutex : protecting the previousOperation global variable, used to determine whether we should perform local cloning of the QueryTable or not.
- queueEmptyCond condition variable : checks that jobQueue is no more empty.

MatchDocument Multithreading:

- resultLock mutex : protecting the resultList global struct, where the results used in GetNextAvailRes are stored.

- resEmptyCond condition variable : checks if resultList is empty. If it empty, then GetNextAvailRes should wait, until all tasks are executed.
- unfinishedDocs global variable : It is incremented when a MatchDocumentJob is assigned to a thread and it is decremented when the job finishes. In this way, we check that no MatchDocumentJob is left unfinished and so, the GetNextAvailRes function can begin.

StartQuery/EndQuery Multithreading:

- entrylistLock mutex : protecting the entrylist global struct.
- unfinishedQueriesLock mutex : protecting the unfinishedQueries global variable.
- unfinishedQueriesCond condition variable : checks if unfinishedQueries is equal to zero, which means that all StartQueryJobs have been executed and so, MatchDocument or EndQuery can begin.
- queriesToDeleteLock mutex : protecting the queriesToDelete global variable.
- queriesToDeleteCond condition variable : checks if queriesToDelete is equal to zero, which means that all EndQueryJobs have been executed and so, DestroyIndex can begin.
- exactLock mutex : protecting the global entry hash table used for exact match.
- editLock mutex : protecting the edit index.
- hammingLock 27 mutexes : protecting all hamming indices.
- bucketLock mutexes : a mutex per Query Table bucket, in order to lock only the specific bucket each time and not the whole hash table.
- unfinishedQueries global variable : It is incremented when a StartQueryJob is assigned to a thread and it is decremented when the job finishes. In this way, we check that no StartQueryJob is left unfinished and so, the MatchDocument and EndQuery functions can begin.

- queriesToDelete global variable : It is incremented when an EndQueryJob is assigned to a thread and it is decremented when the job finishes. In this way, we check that no EndQueryJob is left unfinished and so, DestroyIndex can begin.

In order to test that our program works well always and not by chance, but also in order to present more accurate time results, we built a bash script that runs the program 100 times, checks that it runs as expected and finds the average time for a run.
These are the average time results taken from running the program 100 times for each multithreading mode on a University machine (Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz with 4 cores):

Locking the corresponding bucket when adding to the Entry Table and using 5 threads:

mode	milliseconds
Sequential	1226
MatchDoc	**882**
StartQuery	1374
EndQuery	1304
MatchDoc-StartQuery	895
StartQuery-EndQuery	1320
MatchDoc-EndQuery	892
multithreading all	899

Locking the whole Entry Table and using 5 threads:

mode	milliseconds
Sequential	1216
MatchDoc	**874**
StartQuery	1367
EndQuery	1298
MatchDoc-StartQuery	896
StartQuery-EndQuery	1316
MatchDoc-EndQuery	892
multithreading all	902

Paradoxically, we get slightly better results having a mutex for the whole entry table instead for each bucket. Now, using diffent numbers of threads:

mode	2	3	4	5	8	16
MatchDoc	926	**809**	849	874	893	913
StartQuery	1351	1379	1376	1367	1369	1369
EndQuery	1286	1307	1305	1298	1293	1306
MatchDoc-StartQuery	903	843	858	896	915	934
StartQuery-EndQuery	1300	1327	1320	1316	1313	1323
MatchDoc-EndQuery	928	833	861	892	912	925
multithreading in all	909	846	864	902	909	923

It seems that the parallelization of only the MatchDocument gives us the best results. That is probably due to the fact that this was the most time-consuming task in the first place. All in all, we observe that after parallelization the execution time has decreased from around 1.200 milliseconds to around 800 milliseconds, which is a reduction of 33.3%.

Furthermore, using the time command we got the following results, concerning the cpu usage, using 4 threads and parallelizing only the MatchDocument part:

user time	1.878
system time	0.444
real time	0:00.86
total CPU time	2.322
CPU percentage	268.6%

Finally, we experimented with the helgrind and drd tools and fixed any issues that occured, in order to ensure that there are no data races whatsoever, and the program functions well.

6 Conclusion

During this assignment, we had the opportunity to do research on plenty of concepts and technologies. We familiarized with Unit Testing, Version Control, Multithreading, which all helped us build quality software, efficient and maintainable and realize the importance of creating solutions to everyday problems through Information Systems.