Πανεπιστήμιο Κρήτης

Τμήμα Επιστήμης Υπολογιστών

HY463 Συστήματα Ανάκτησης Πληροφοριών

Εξάμηνο: Άνοιξη 2022

**Γραπτή Αναφορά Έργου**

**BioMedic Indexer**

Στοιχεία Φοιτητών

|  |  |
| --- | --- |
| Ονοματωπώνυμο | Manos Chatzakis |
| ΑΜ | 4238 |
| Email | csd4238@csd.uoc.gr |

**Table of Contents**

[1 Introduction 3](#_Toc471646822)

[2 Implementation 3](#_Toc471646823)

[3 Experimental Evaluation 3](#_Toc471646826)

[4 Conclusion 3](#_Toc471646827)

[5 references 3](#_Toc471646828)

# Introduction

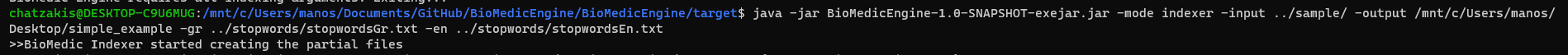
This project is the implementation of a BioMedical Search Engine over a biomedical document collection of 5GB.

**How to run:**

It is mandatory to execute the **exejar** file, located at **target/** folder.

* **Indexer**: An example is

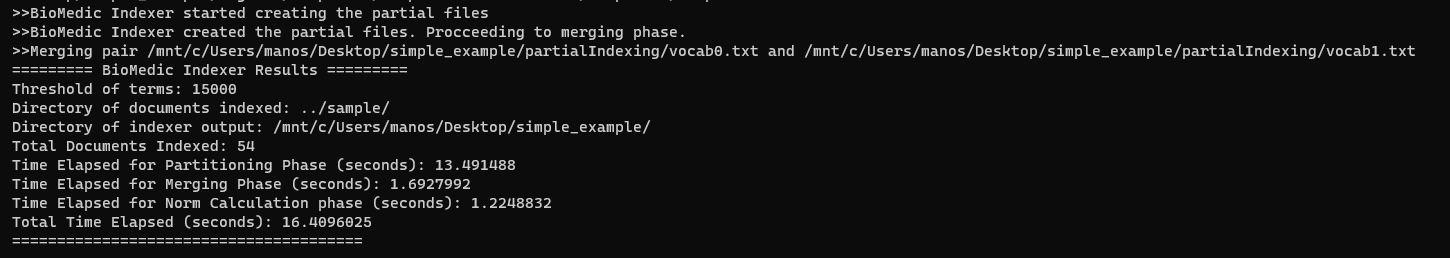
***java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar******-mode*** *indexer* ***–input*** *../sample/* ***-output*** */mnt/c/Users/manos/Desktop/simple\_example* ***–gr*** *../stopwords/stopwordsGr.txt* ***-en*** *../stopwords/stopwordsEn.txt*

**

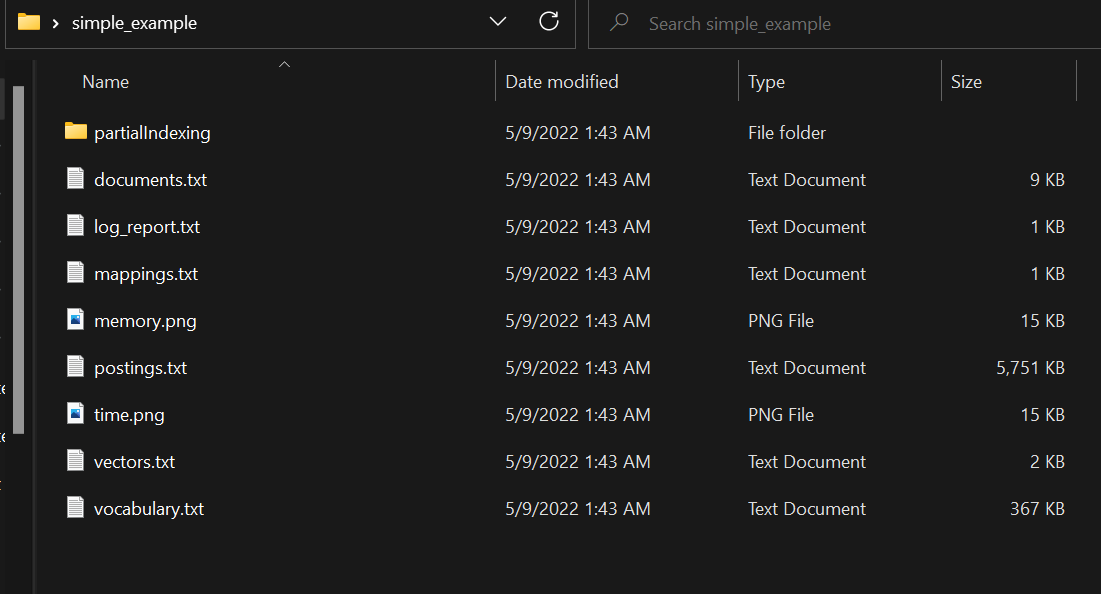
**Important Note 1:** It is mandatory that the output directory has a folder named: “partialIndexing/”, in which the partial files will be stored.

**Important Note 2:** Always put “/” at the end of the directories.

The output of the program should look like:



And the output directory should look like:



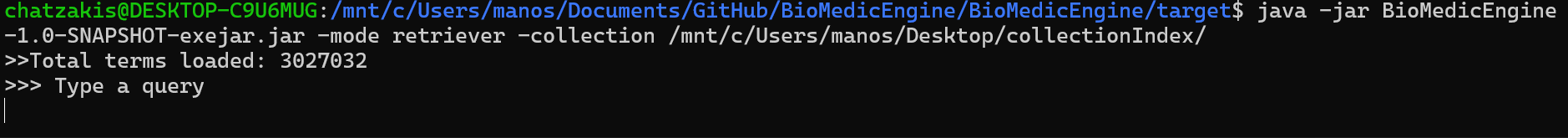
* Retriever: BioMedicRetriever should run from CLI using

**java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar** **-mode** retriever **-collection** /mnt/c/Users/manos/Desktop/collectionIndex/

for the simple retriever which uses the vector model and using

**java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar** -**mode** topicRetriever -**collection** /mnt/c/Users/manos/Desktop/collectionIndex/

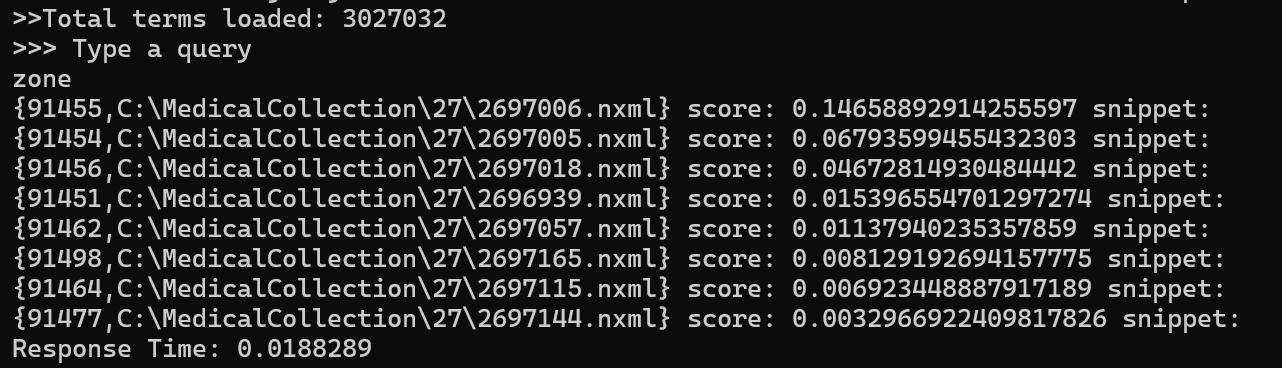
The output of the program should look like this:

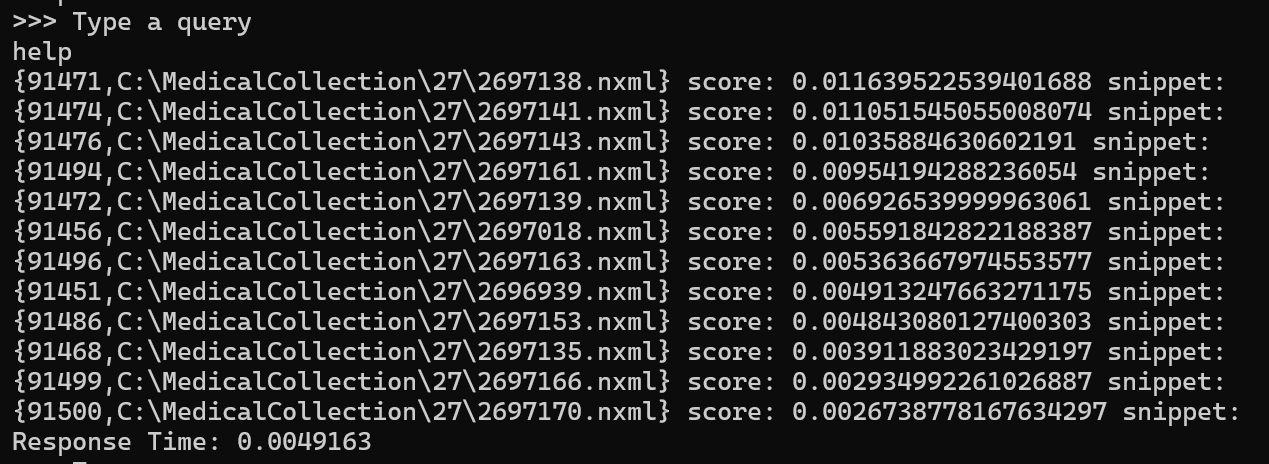


And this for the topic retriever:

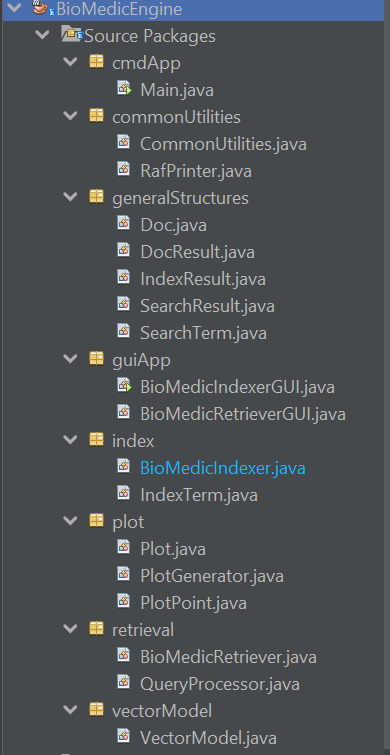
After this, you can start querying the collection. The initialization of the tool might require some time.

Example:





**Project Layout:**

****

# Implementation

In this section, the basic methods used to implemented both parts of BioMedic Indexer (Index Creation and Query Answering are described).

## Index Creation

BioMedic Indexer index a selected directory in three steps: (a) Partial Indexing (b) Partial Merging and (c) Document Norm Calculation.

**Partial Indexing.**

BioMedic Indexer uses a sorted <String, Term> map to store the terms. During the first phase of indexing, the terms and related information are stored into this map. This map stores their df, their occurences per tag etc. The document collection is read sequentially and the contents of each document are added to the map.

When the size of this map gets greater than a threshold TH, the contents of the map are flushed to the disk, creating a pair of partial files, with names “vocabX” and “postX”. These files are stored in a list. Every time these files are flushed to the disc, this list is cleared.

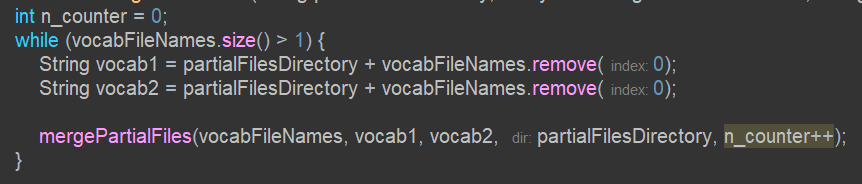
BioMedic Indexer uses a relatively small TH, to be able to run in systems with small memory capacity.

**Merging.**

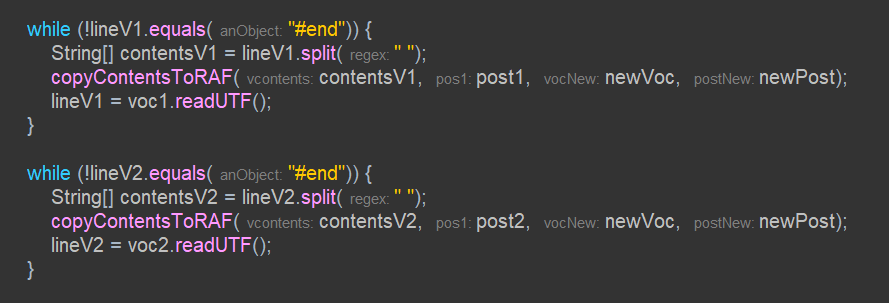
After the partial indexing phase is completed, the files need to be merged. To merge the files, the tool removes two files, and adds the output of merging to the list, till the list size is 1. Then the remaining file is the vocabulary file, and the corresponding posting is the posting file.

All of this files are maintained, traversed etc. using the Java RandomAccessFile API.

Merging Algorithm (Like merging two linked lists☺):

****

****

****

**Document Norm Calculation.**

The norms are calculated in a different file and stored separately. After the completion of the partitioning and merging, we initialize the vocabulary and we keep a map <Integer,Double> which stores the mappings of the document ID with it’s norm. We traverse the terms one time and if a document contains the term we add (TF\*IDF)^2 to the total current norm.

After the traversal, we write the SQRT(map) in a new random access file called “norms” and we save the mappings of document IDs and the map pointers.

## Query Answering

Here, the process of query ansering is described.

**Vector model.**

*[Step 1 – Initializing BioMedic Retriever]:* Given a directory to index, the vocabulary is initialized and kept in memory, while we also load the pointers to the Random Access Files.

*[Step 2 – Getting the relevant documents]:* Given a query, the query processor parses the query using “ “ and finds its terms. Then, we traverse the terms one by one, and for every term present in the vocabulary, we traverse it’s postings and retrieve the documents in a list. This list contains the relevant documents.

*[Step 3 – Finding the norm of the vector]:* The query processor not only parses the query to its terms, but it returns a map of <Term, TF>. Thus, using the TF of the term inside the query and the iDF as it comes from the model, we can calculate the norm of the query the same way we did for the documents. Indeed, terms that are not present in the vocabulary are removed.

*[Step 4 – Find the dot product per vector]:* For every relevant document, we do the following. For every term in the query, we have the queryTF and iDF. Then, we traverse the postings of this term. If the documentID is found, we return the corresponding TF, and so, we calculate the dot product as the sum of (queryTF\*iDF)\*(docTF\*iDF) of every term.

*[Step 5 – Find the score of document]:* Given that we have the dot product and the norms available, the score is (dot product)/(docNorm \* queryNorm).

*[Step 6 – Return the results]:* The documents are stored in a sorted list based on their score. This list is returned, with the time needed to answer the query.

**Vector model with Examination type support.**

*[Step 7 – Support Examination Type]:* The goal of this part is to not only return the documents related to the query, but also try to return documents that also correspond to an examination type. To support this, we do the following. We retrieve the documents related to a query the same way we did previously. Then, we retrieve the documents related to a topic, i.e. using the examination type as the query. As a last step, to return the most related documents, we return the **intersection** of these two sets, using the score of the queries that was given from the vector model. This way, the engine filters out the documents that are unrelated to the examination type, and keeps the ones that surely contain some information related to this type. Another approach could be to assign the score they had as it came from the query of the examination type.

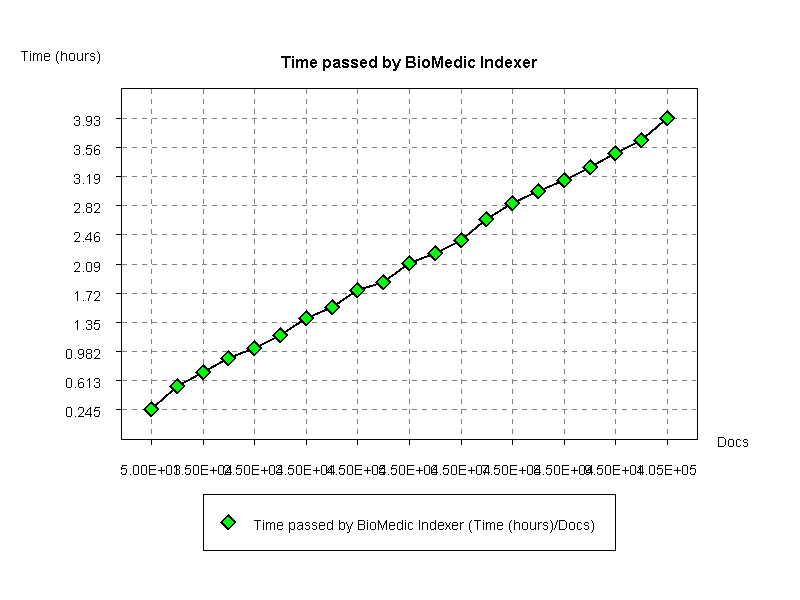
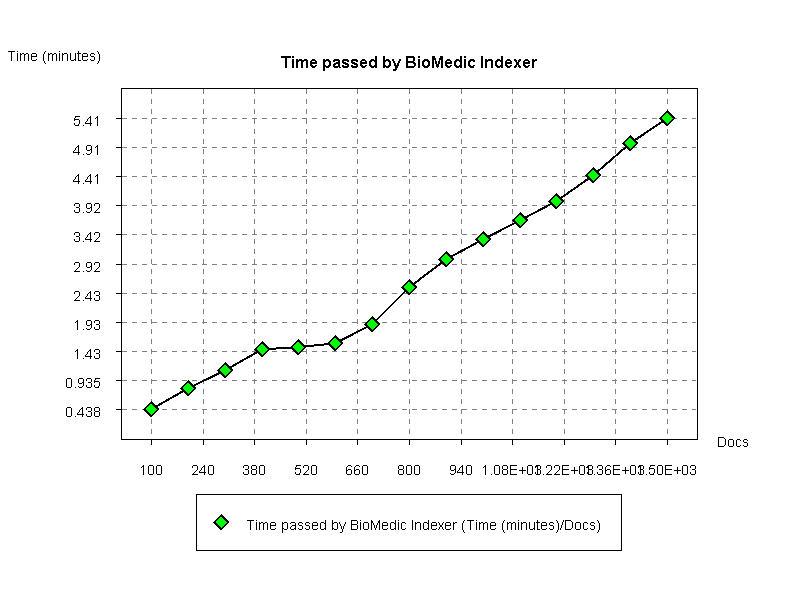
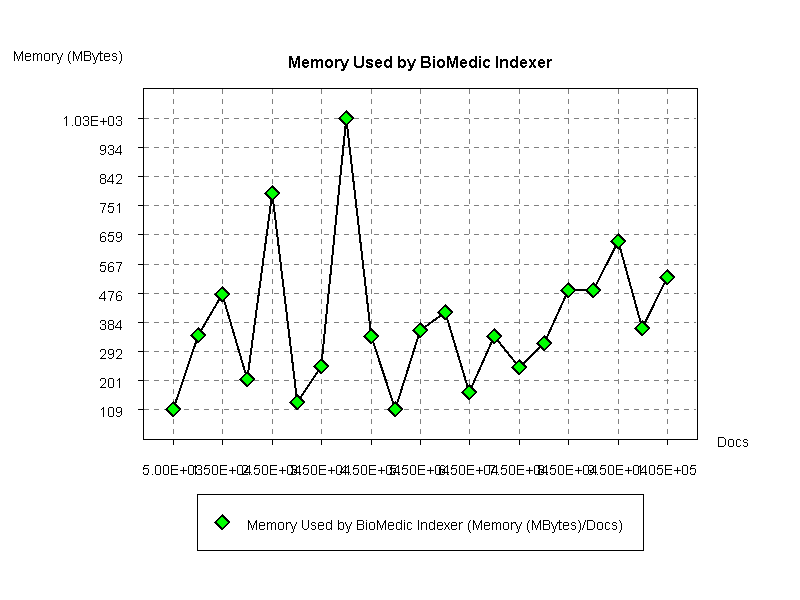
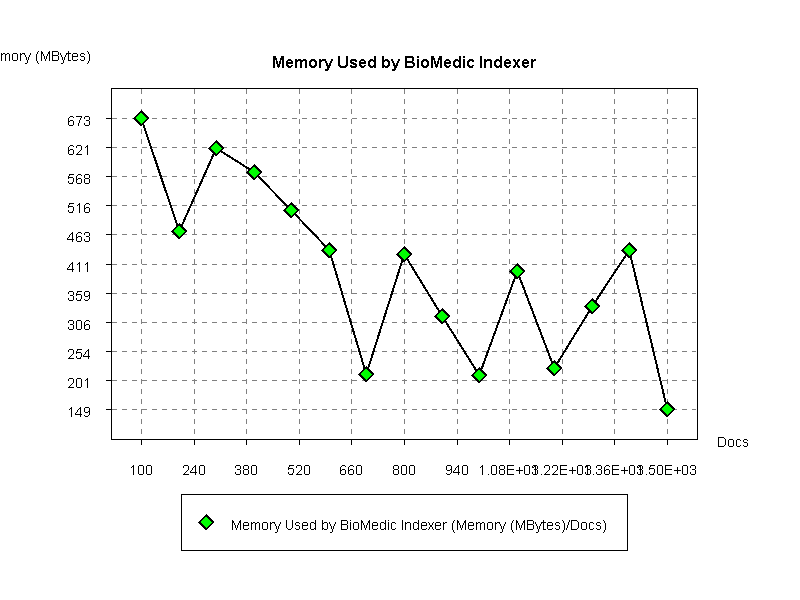
*Other methods for this problem that did not work.* Generally, given that we want to assign better scores to the documents that are related to a specific examination type could be to implement a weighting method of the terms, so that that the documents that contain the word “type = {test, diagnosis, …}” could get a better score. This method did not give proper results.

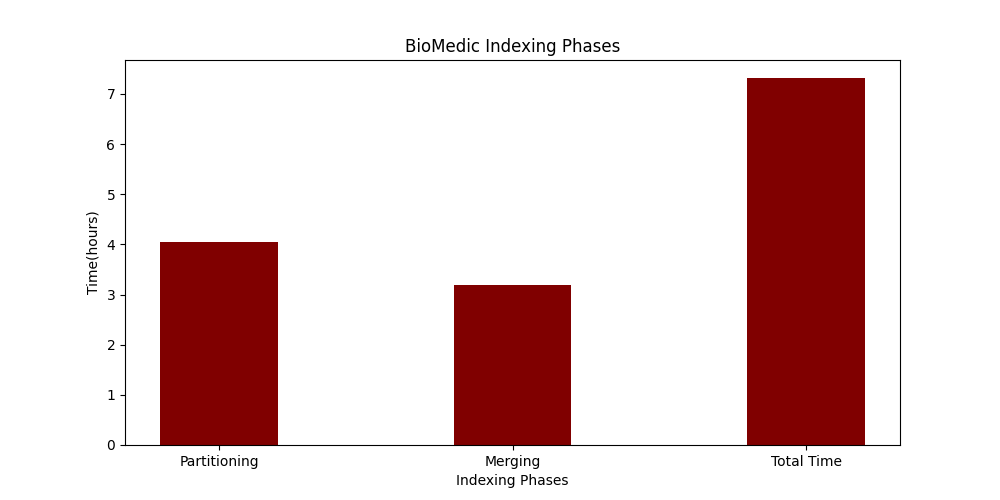
# Experimental Evaluation

The experiments contacted on a machine of 8GB Memory, 256GB SSD NVMe Disc, and a 4-Core (8 Hyperthreads) i5 8th-gen CPU, running Windows 11.

## Index Creation Evaluation

The evalution of the indexing phase is presented in two graphs: A graph showing the memory usage correlated with the document count, and a graph showing the total time passed correlated with the document count. For both of them we show two versions: One for the whole indexing process and one for the indexing of a subset of the collection.





## Query Answering Evaluation

For query answering, we show the total response time needed for the queries created from the files of “topics.txt”.

# Conclusion

This report is a presentation of a vector-model-based Search Engine. It contained a basic tutorial related to “how to run”, a basic explanation of the architecture, a presentation of the methods that where used to implement the engine and the experimental evaluation.