Πανεπιστήμιο Κρήτης

Τμήμα Επιστήμης Υπολογιστών

HY463 Συστήματα Ανάκτησης Πληροφοριών

Εξάμηνο: Άνοιξη 2022

**Project Report**

**BioMedicEngine**

Phase Β

Student

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

***Changelog from PhaseA***

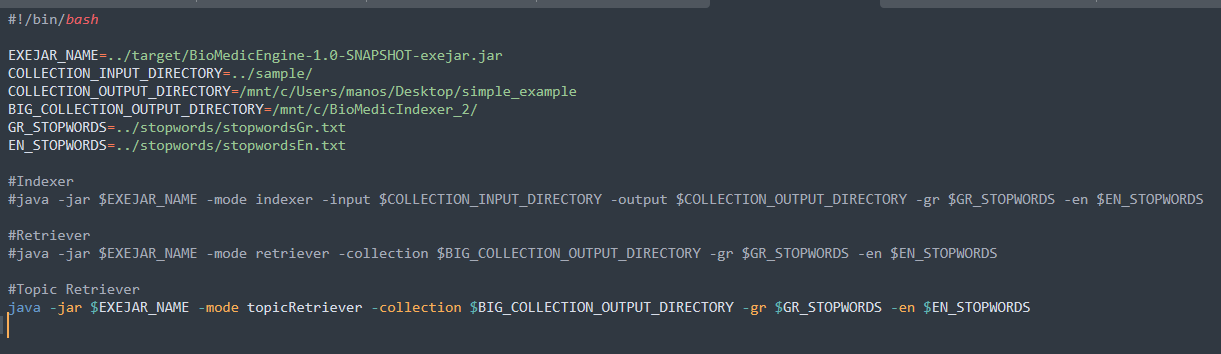
* Minor bug fixes for index creation, in merging algorithm
* Minor bug fixes for query answering using the Topic BioMedic Retrieval
* Performance Optimization in the BioMedic Retriever Package
* Code Clean-Ups
* Updated diagrams for experimental analysis
* Updated report
* Implemented Query Answering Quality Evaluator package which contains methods to
  + Save the top-1000 results from the topics.xml file in a result file
  + Evaluate these results using qrels by implementing BPREF.

# Introduction

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

**How to run:**

BioMedicEngine has a bash script to easily configure and run the engine through command line, in the bash folder of the project.



To run, you need to configure the directory variables and comment/uncomment the corresponding run mode. Then, the engine is initialized using “./run.sh”.

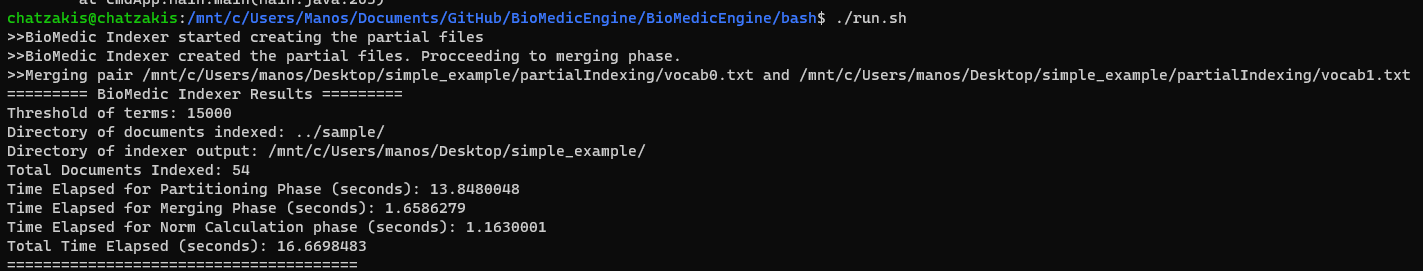
**Examples:**

Creating an index: Initialize the engine using the aforementioned script by configuring the variables and the mode, to index the corresponding folders.

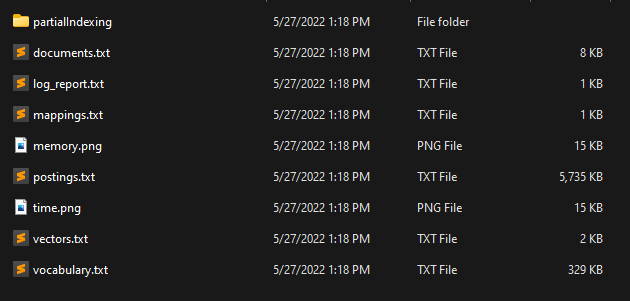
**Important Note 1:** It is mandatory that the output directory has a folder named: “partialIndexing/”, in which the partial files will be stored.

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

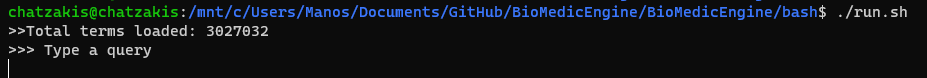
The output should look like this:



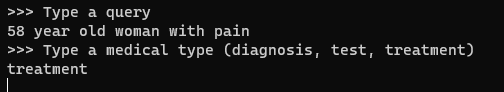
The directory should contain the following files:



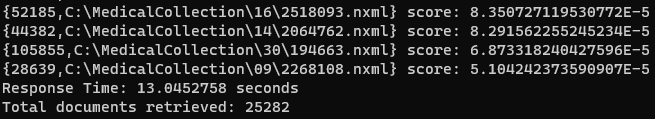
Quering the engine: Initialize the engine using the script. The output is this below.



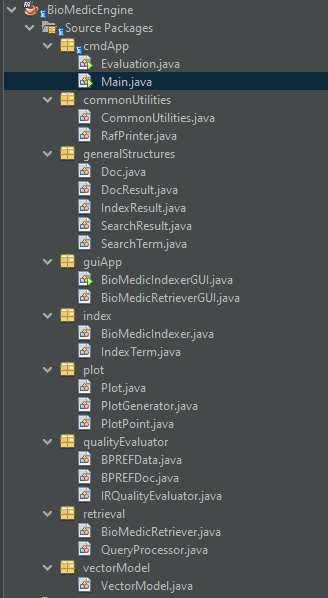
Then, you can start applying queries and their topic:



BioMedic Engine responds:



**Project Layout:**

****

**About the architecture:**

Package index: This package contains all classes needed to Index a directory.

Package retrieval: This package contains all classes needed to perfrom query answering (simple or topic).

Package cmdApp: This package contains the CLI application of the BioMedicIndexer. It also contains the automation methods for quality evaluation, in the Evaluation class file.

Package generalStructures: This package contains the general structures needed by (most of) all other packages, such as “Document” etc.

Package guiApp: This package contains the GUI applications (not finalized).

Package plot: This package contains methods to create the plots (used in the report of the index results).

Package vectorModel: This package contains all methods needed to calculate norms, TF\*iDF arrays etc.

Package qualityEvaluator: This package contains all the methods needed to automate the quality evaluation process and calculate the per-topic-BPREF.

Package commonUtilities: All utility functions.

# 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.**

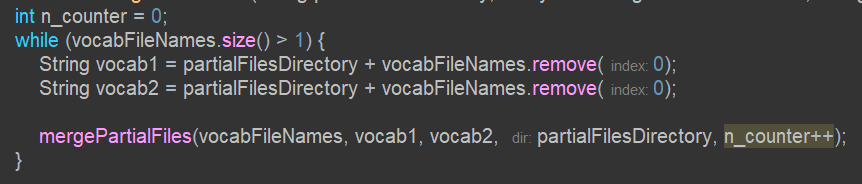
After the partial indexing phase is completed, the files need to be merged. To merge the files, the tool removes two files from the list, and adds the output of merging to the list, till the list size is 1.

This list contains the names of the partial vocabularies files, and the corresponding posting file is found by replacing the part “vocab” with “post”. This is why the directory path should not contain words such as “vocab” and “post” to avoid such mistakes.

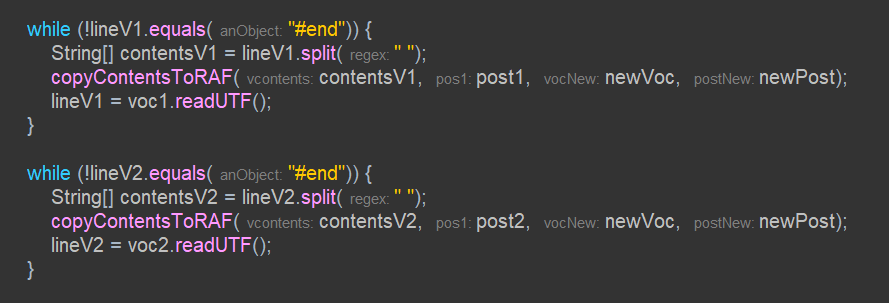
Then the remaining file is the vocabulary file, and the corresponding posting is the posting file.

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

Merging Algorithm (Like merging two linked lists☺): We create the new posting and vocab file keeping the the lexicographical order the same. For every term added to the file, we should also merge their postings, in an ordered way using the document IDs, as seen in the algorithms above.

****

****

****

**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 its 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.

**Query Processor.**

The query processor is a class that parses a query, removes the stop points (eg. “.”, “,” “( )” …) and returns a map that consists of the query terms and their TF. Also, it can load files of stopwords just like the index creation, using the same files and method calls.

## Checking and Debugging

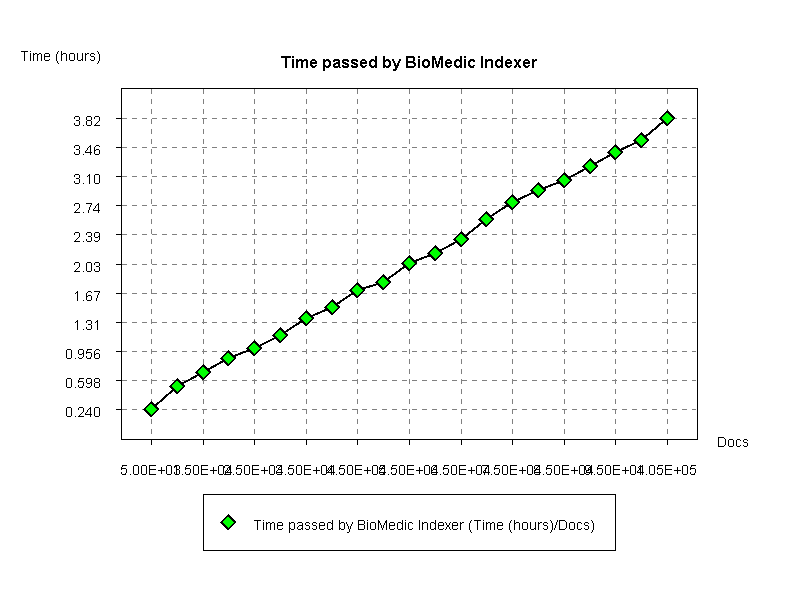
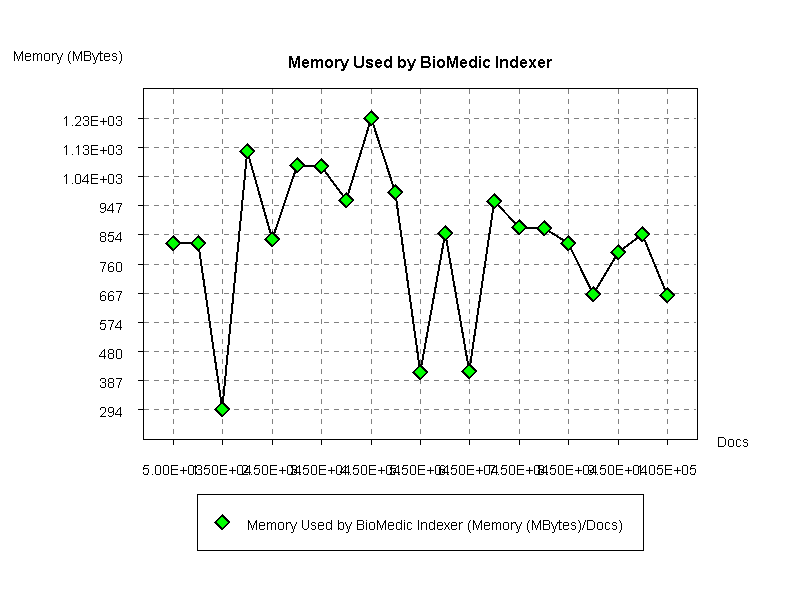
To check if the merging and qa are working fine, we did the following checks. First of all, after the indexing of the big collection, we checked if a norm is calculated for every document ID. In addition, we printed the biggest parts of the RandomAccessFiles and we checked that the contents are stored properly. Regarding query answering, we made some simple queries and we checked if the terms appear in the resulting files. We also made plenty of queries (randomly) to check that the system isn’t crashing. Note that an extended evaluation of the results of this (and any) system is the topic of the next phase of this project.

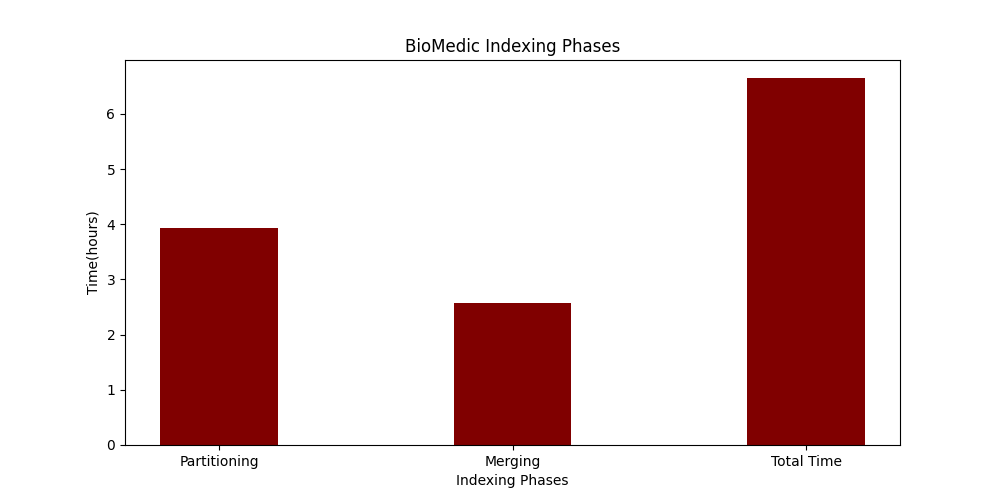
# Experimental Evaluation

The experiments contacted on a machine of 16GB Memory, 1ΤΒ SSD NVMe Disc, and a 8-Core (16 Hyperthreads) 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 the partitioning phase of the algorithm.

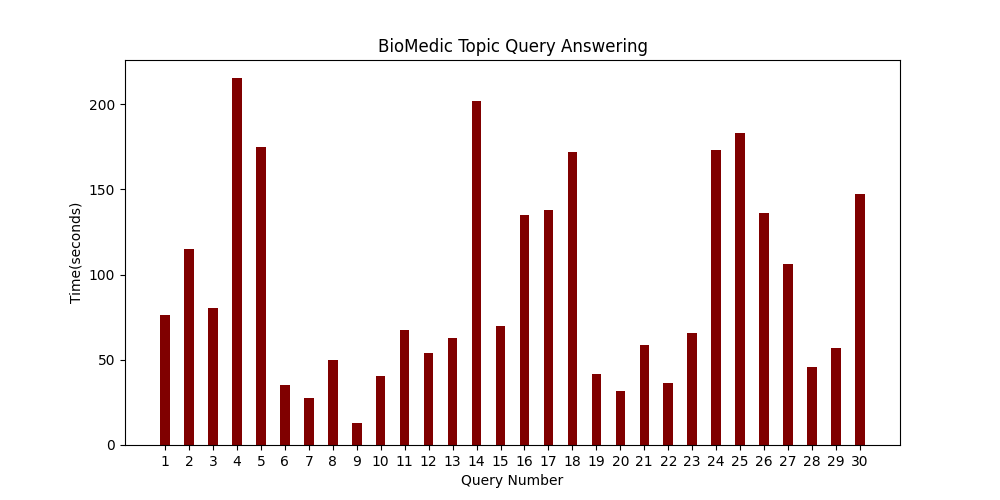




Note that the total time needed to index a directory is also based on the threshold we choose. Choosing a smaller threshold to maintain the total memory used in small levels (see corresponding figures) could be beneficial as the program can run in any machine.

## Query Answering Evaluation

For query answering, we show the total response time needed for the queries created from the files of “topics.txt”, using the summaries of these queries. The Results can bee seen in the graph below.



This evaluation is performed automatically from the system, using statistics gathering and analysis, and can be reproduced easily.

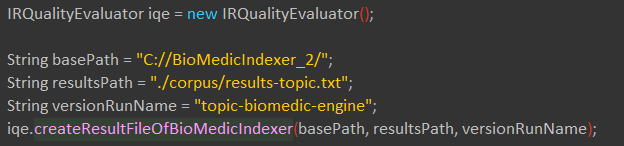
# Quality Evaluation

BioMedic Indexer comes with a Quality Evaluator package, that calculates the results of a set of queries, as given from the BioMedic Indexer, and then calculates some evaluation metrics, e.g. BPREF.

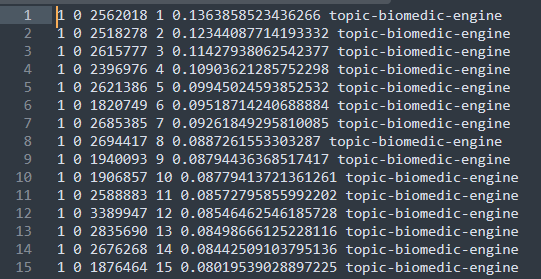
This process is automated and it consinsts of two phases: (a) creation of the result file and (b) the evaluation of the results.

## Creation of the result file

As a first step, a result file that holds the top-1000 answers from specific medical topics, sorted by their score, one per line:



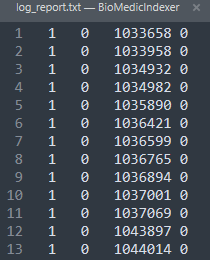
The output of this process is a file of this form:



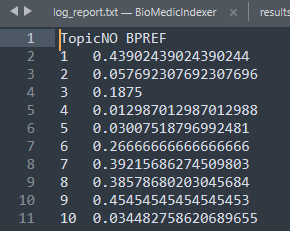
This file is later used to evaluate these results.

## Evaluation of the results using external knowledge

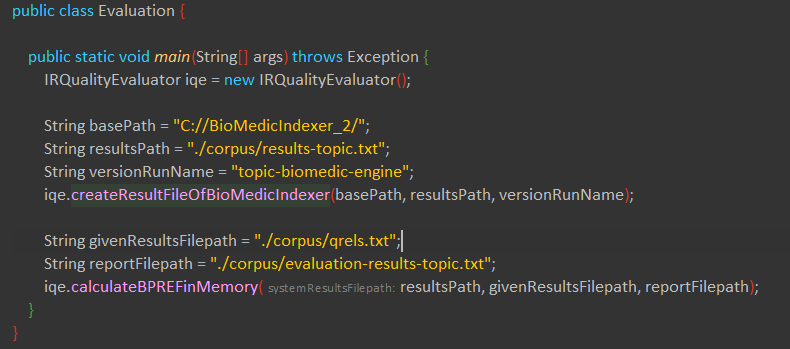
The results are evaluated using an external file (“qrels.txt”) which is a TSV file of the form:



Holding information about which files are relevant to the medical topic and which are not. The data of this file are loaded in-memory, then the data saved in the result file of the previous phase are also loaded and BioMedic Indexer calculates the per-topic-BPREF based on these results and produces a file of the following form:



Again, the process of the BPREF-calculation is automated and can be reproduced for different engines, algorithms etc. An outline of this process in the code is given in the following image:



# Conclusion

This report presents the outline of the work for the creation of a BioMedical Search Engine. It contains basic information about how to use the engine and the architecture, while the algorithms used for index creation and query answering are presented and explained. Moreover, a modification of the vector-model is presented to be able to retrieve documents that are more relevant to a specific topic (treatment, diagnosis, test) apart from the classic vector model. Also, an experimental evaluation for both indexing and query answering is shown, while the project contains a quality evaluator module that is used to evaluate the results of the engine.