HY463 – Information Retrieval Systems **Due: July 16th**

Project Part (A & B)

Instructor: Panagiotis Papadakos
Names: Mahfoud AMHIYEN – Myrto SALVARAKI

System requirements

The indexing and querying tests were performed with a computer with 8GB of RAM and a regular HDD disk. We also increased the heap size available to Eclipse to 3GB.

In addition, the tests were performed on corpora of different sizes, i.e. one corpus containing only one file, another containing 35 files and the last one containing 100 files, equivalent to 1.6 GB.

During the different tests, we encountered a "NullPointerException" due to a line break and a special character in the some files. So we decided to take into account only words of the Latin alphabet, which resulted in an increase of indexing speed.

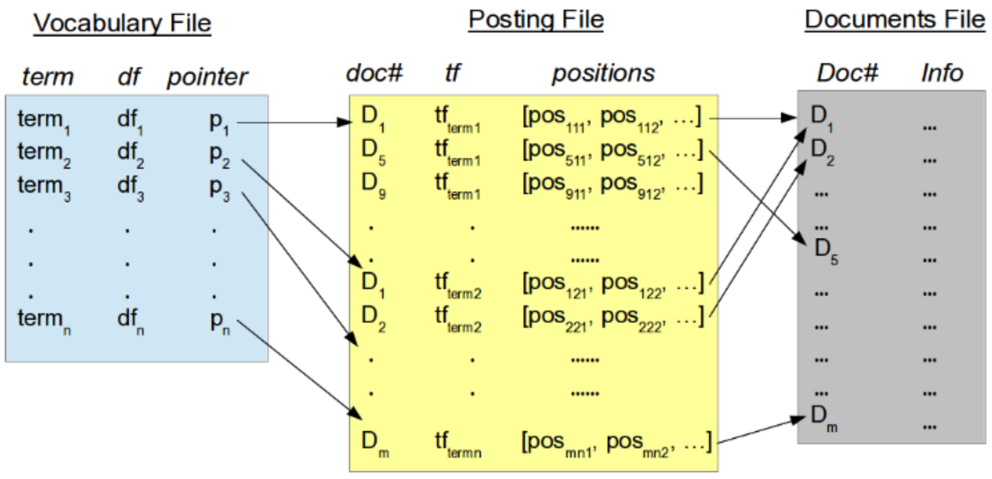
We will discuss the test results in more detail in the section 2.

We also used Pangaia to test our system. First we ran the tests on the HDD and then when we made sure that the test was working well, we ran it on the SSD.

1 Indexation process

For indexing purposes, we considered it appropriate to index only the title and abstract for each document, which represents the most interesting textual information, and then make queries.

Based on image 1, we deduce that the vocabulary file is depended to the posting file which is depended to the documents file. This is why we decided to create a new class called InvertedIndex, that will store the df, the tf and a list of pointers to the document file containing the word. Plus, we kept the idea of using a hashmap to contain the vocabulary, replacing the int by the InvertedIndex class. So we'll have the word as the key of the hashmap and InvertedIndex as the value.



Vocabulary File
term df pointer
term1 df1 p1
term2 df2 p2
term3 df3 p3
termn dfn pn
Posting File
doc# tf positions
D1 tfterm1 [pos111, pos112, ...]
D5 tfterm1 [pos511, pos512, ...]
D9 tfterm1 [pos911, pos912, ...]
D1 tfterm2 [pos121, pos122, ...]
D2 tfterm2 [pos221, pos222, ...]
Dm tftermn [posmn1, posmn2, ...]
Documents File
Doc# Info
D1
D2
D5
Dm

Figure 1: The inverted index

We start the indexing process, by reading all the files of the corpus one by one, creating a list of S2JsonEntryReader. Then, we browse the S2TextualEntry elements of the list. We write the content of the S2TextualEntry in the document file, which returns the position for the next document to be written. We decided to consider the vocabulary terms by spliting the S2TextualEntry's entry by blank space and considering only words of the Latin alphabet. We go through the split word array, we clean the word, we calculate the tf. We also check if the hashmap already contains the word, if so, we update the tf and the list of pointers to the document file. If not, we add the word to the hashmap, with a tf of 1 and the pointer to the document file.

Then comes the process of writing the vocabulary and the posting file, which consists in browsing the hashmap, retrieving the tf, the df, the word and the positions to the document file. We write the tf and the list of pointers in the posting file (method writePosting, which returns the pointer of its current position in the file), And finally, we write the word, the df and the pointer to the posting file in the vocabulary file.

1.1 Partial indexing

Partial indexing is used when the corpus size is larger than the available RAM memory. The user can decide whether to use partial indexing or normal indexing. However, we strongly encourage the user to use partial indexing because it is more optimal, even for small corpora.

For partial indexing, we perform the following tasks: - For each file, we read the different articles, extract the words to index, and save them in a hashmap until it reaches a certain number of lines. After reaching this line limit, we write the data (the word, the tf, the idf, the list of pointers) in a file, ”dumping in” the partial index. Then we increase the number of partial indexes by one. The process continues until all the files have been indexed. When we have gone through all the files in the corpus, we continue with the merging process, which consists of combining two files into one. The idea is to browse x lines of file one, keep it in memory and compare it with the x lines of file two. If both files start with the same word, we merge the data, i.e. we add the tf, idf of both files, but also we combine the pointer lists of the documents file. We proceed this way until we get a single final file. This process might take a long time, depending on the number of lines in each file. That’s why a lot of tests were conducted on our machines to find the optimal number of lines in order to reduce the merging time. Finally, when the final partial file is written, we extract and write the data in the posting and vocabulary files. This process is the fastest of the 3 steps composing the partial indexing.

2 Results and discussion

Indexing a single file containing 10,000 documents, the execution time is on average 2s to 3s, which depends on whether the PC processor is also used by other applications or not. This seems relatively low considering the number of documents to be indexed.

Indexing the 1.6 GB collection the execution time, with the equipment described above, is 705 seconds (11 minutes and 45 seconds). This leads us to believe that the indexing time is constant. To be sure, we decided to calculate the indexing time of each file of the corpus. The graph 2 shows the indexing time pf each file of the collection, and thus shows that it is quite constant.

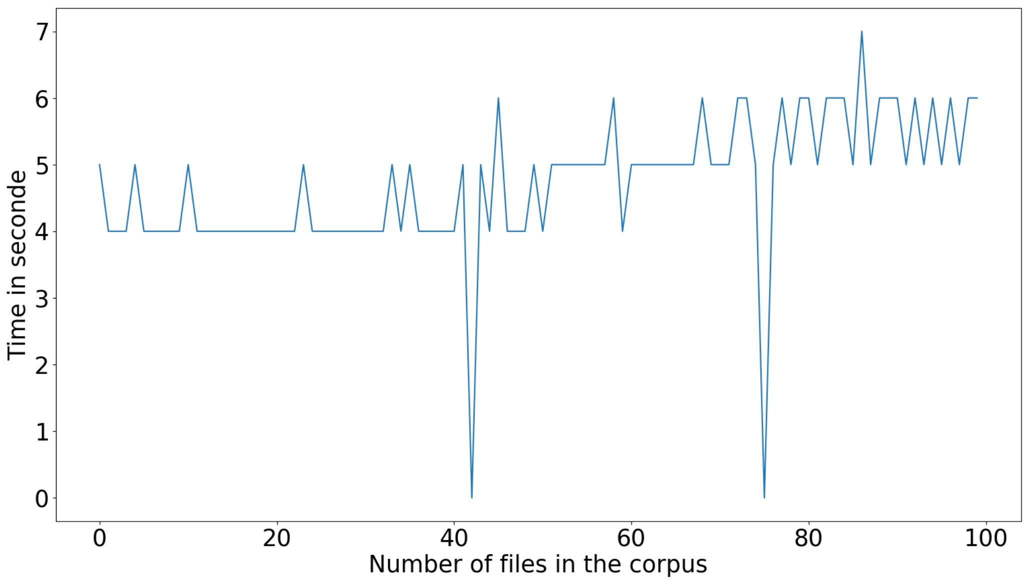


Figure 2: Indexing time for each corpus file

The two peaks at about 40 and 80 can be explained by the fact that both files were very small, only a few

KB in size, so the writing process in the document file is almost instantaneous.

Thus, we have managed to reduce the indexing time, which was originally linear, to around 5 minutes for the entire indexing process. The indexing process takes around 320 seconds (5 minutes and 20 seconds) and the merging and the creation of the vocabulary and posting files takes around 30s.

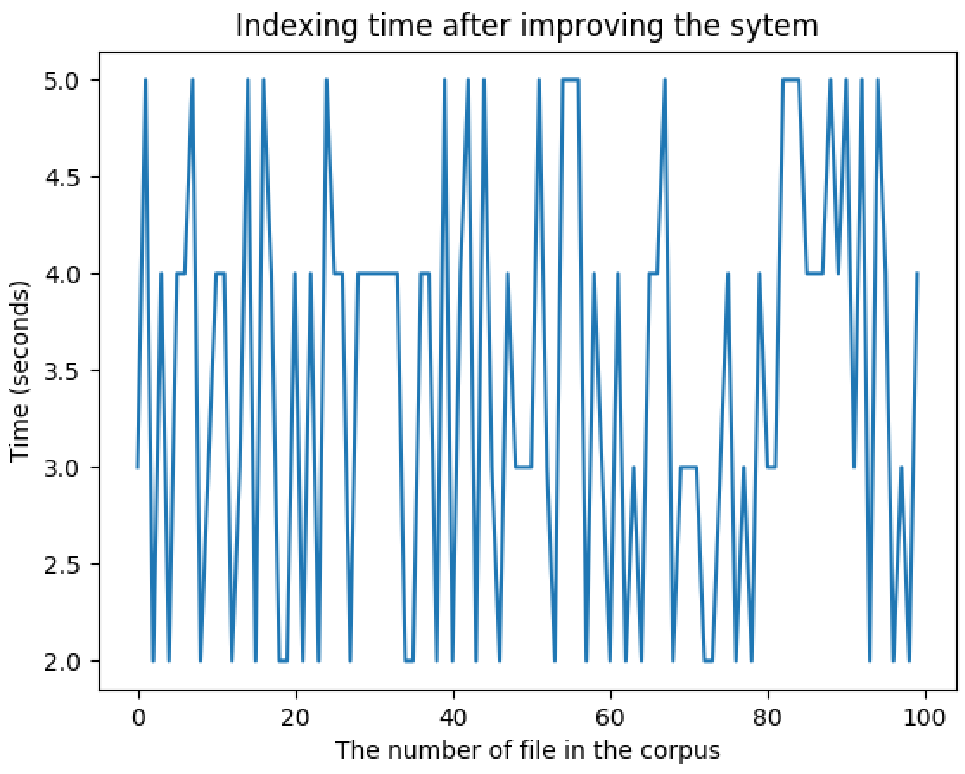


Figure 3: Improved system

We were also interested in the size of the index, counting the three files, i.e. the document file, the vocabulary and the posting file. For a 16.0 MB corpus file, the index size is 10.6 MB, which is about 62% of the initial file. In addition, the index size for the entire corpus, 1.6 GB, is 1 GB.

On Pangaia, the indexing process, using 750k publications per partial, takes 22.855 seconds (6 hours, 20 minutes and 55 seconds) and the merging process takes around 40% of the time. The index size is about 89 GB (Vocabulary 0.9 GB, Document 4.9 GB, Posting 83 GB)

If an indexing process improvement is needed we would start with the merging process.

3 Query time

After having indexed all the articles in the corpus, we start searching for words and phrases. At first we did these tests on our computers and when the search times seemed good, we performed the tests on Pangaia. However, when tests on Pangaia were done, we realized that the system is very far from being in real time, as it took up to an hour to perform a simple query. This forced us to revise our way of doing things, which was: reading the pointer to the posting file for each query word, then reading the line in the document file corresponding to each article in the list containing that word and then extracting the necessary data.

Despite a few attempts to improve the process, we have not managed to do better than our algorithm mentioned above, because we want to keep the filename, whose string size we don't know.

For our tests, ran with the 1.6 GB corpus, we retrieved a list of words and phrases :

- Lydia
- piazza
- sensitivity

- electroporation
- caffeine
- primitive Mexico

For each of these queries, we ran a search for Okapi and VSM. And the result was this:

Query	VSM Time (seconds)	Okapi Time (seconds)	Number of Results
Lydia	1	2	192
Piazza	1	1	72
sensitivity	900	900	1520
electroporation	3	3	250
cafeine	0	0	2
primitive Mexico	33	33	1975

Table 1: The query times

This shows that our system is not a real time system, so we still need to improve the query time. The query time increases in function of the number of results, even if it is a phrasal query or not.

Because the query "Sensitivity" is very common in the articles, it takes a long time to find all the articles containing the query.

For the phrasal queries, the idea is to split the query by whitespace and then search for each word in the query. Given our algorithm it may take a long time to find all the results for a given phrasal query.

4 Page rank

The basic principle is to attribute to each page a value (or score) proportional to the number of times a user browsing the graph would pass through this page by randomly clicking on one of the links appearing on each article. Thus, an article has a PageRank all the more important as the sum of the PageRanks of the pages pointing to it is large (including itself, if there are internal links).

Based on this principle, we have implemented our page rank algorithm, which consists of saving in a Hashmap the article identifier and the pages that point to this article. Then, for each article A, adding the sum of the pages pointing to it and also the score of the pages pointing to the pages that point to article A. However, when the sum of the final score is very small or there are no other articles pointing to article A, the final score is 0.

It takes about 58 seconds for the 1.6 GB corpus to go through and calculate all scores.

5 Query expansion

In order to improve the user's search, we have also implemented some query expansion algorithms.

5.1 Using Wordnet

WordNet is a lexical database, its purpose is to index, classify and relate in various ways the semantic and lexical content of the English language.

The atomic component on which the entire system is based is the synset (synonym set), a group of interchangeable words denoting a particular meaning or usage. The WordNet lexicon is separated into four major lexical super-categories: nouns, verbs, adjectives, and adverbs. Nouns are thus classified into a comprehensive and accurate category system with several levels of nesting.

We provide a function called getAllSynsets() which gives us all the synomyns of a given word. In this example we ask the database for the synonyms of the words :

- female = [female, female, distaff, female]
- tail = [tail, fag end, tail, tail end, tail, tail end, buttocks, nates, arse, butt, backside, bum, buns, can, fundament, hindquarters, hind end, keister, posterior, prat, rear, rear end, rump, stern, seat, tail, tail end, tooshie, tush, bottom, behind, derriere, fanny, ass, tail, shadow, shadower, tail, tail, tail assembly, empennage, stern, after part, quarter, poop, tail]

- person = [person, individual, someone, somebody, mortal, soul, person, person]
- book = [book, book, volume, record, record book, book, script, book, playscript, ledger, leger, account book, book of account, book, book, book, rule book, Koran, Quran, al-Qur'an, Book, Bible, Christian Bible, Book, Good Book, Holy Scripture, Holy Writ, Scripture, Word of God, Word, book, book]
- actor = [actor, histrion, player, thespian, role player, actor, doer, worker]
- retrieval = [retrieval, retrieval, recovery, retrieval]
- information = [information, info, information, information, data, information, information, selective information, entropy]

The results most of the time are nouns and the synonyms are very close to the ones we can find on the usual dictionaries.

Also, as we can see, there is a lot of duplicates for some queries, so we have to filter to avoid them.

We also provide another function called getDerivedAdjective(), which gives us the derived adjective to a given word.

5.2 Using Word2vec

Word2vec uses a neural network to find synonyms from an input query. The output of Word2vec's neural network is a vocabulary in which each element is associated with a vector, which can be entered into a deep learning network or simply queried to detect relationships between words.

We provide a function called getAllNearest() which gives us N (by default N = 10) synomyns of a given word. In this example we ask the database for the synonyms of the words :

- female: [male, adult, young, women, woman, adults, girls, individuals, teenage, age]
- tail: [tails, legs, nose, elongated, fins, wings, curved, feathers, beak, snout]
- person: [someone, every, anyone, actually, knowing, man, one, same, woman, another]
- book: [books, story, novel, writing, published, biography, author, wrote, written, titled]
- actor: [starring, starred, actress, comedian, filmmaker, screenwriter, comedy, film, entertainer, actors]
- information: [source, data, sources, provided, documents, search, web, provide, knowledge, intelligence]
- retrieval: [archiving, visualization, synchronization, real-time, annotation, authentication, mapping, workflow, web-based, decoding]

This method gives us pretty much the same results as WordNet. Because we restrain the algorithm to give us only the Top 10 results, it may seem that Word2vec is less efficient than WordNet at finding synonyms. But in reality, Word2vec returns a list of words in ascending order of scores, so the word closest to the query will be first and the next ones will follow.