NYU Tandon School of Engineering
Computer Science and Engineering
CS 6913, Fall 2024

October 3, 2024

Assignment #2 (due October 21)

In this assignment, you are asked to write a small search system that creates an inverted index structure from a large set of text passages, and then returns ranked search results when a user inputs a query. This assignment may be done in groups of two students, though you may do it alone if you prefer to do so.

Your resulting system should consist of (at least) three separate executables, one that parses the data and creates sorted files of index postings in an intermediate format, one that merges the files into a final compressed inverted index, and one that allows a user to input a query and then returns a ranked list of results based on the BM25 ranking function. All data produced by the first and second executable should be written to files on disk to be read by the next executable. You may choose from a number of different programming languages, and may also use different languages for different parts. However, your solution should be fast as well as memory- and I/O-efficient. Most students use C/C^{++}, Java, or Python for this assignment – if you use Java or Python, be sure you know how to write efficient code, manipulate binary data, and do efficient I/O to disk. Other recommended languages include Rust and C#. If you plan on using a language other than these five, please let the instructor know it advance.

Your system should index and search the MS MARCO passage-ranking dataset provided by Microsoft, which consists of 8.8 million passages. The dataset is already in a preprocessed format, so it should be fairly easy to read and parse it, and you may use suitable parsing libraries. Note that the data may be larger than memory size, and thus you must use I/O-efficient methods for index construction as discussed in class. You should use your own machine, most likely your laptop, to run the system. Note that there will be a demo for each group, so make sure you can demo the final search system to the TAs on campus.

After the first two of the three parts, you should have created a complete search index on disk, consisting of the actual inverted lists, the lexicon storing for each terms where the corresponding inverted list is located, a page table allowing you to convert document IDs back into document names (such as URLs), and if needed additional files storing meta information for each block in your index, such as the last document ID in each block and the block sizes. The total index should consist of a small number of files, say three to five – do not create a file for each inverted list or for groups of inverted lists. For full credit, the inverted lists should be stored in a blocked, binary, and compressed format (but you may want to have an option to use ASCII format during debugging).

When your query processor starts running, it may read some of these files into main memory and place them into suitable data structures. However, your query process should not load all inverted lists into main memory! You may choose to support index caching, either static or dynamic, by placing a subset of the inverted lists into a fixed-size cache (say a few hundred MB), but this is optional.

Given a user query, your query processor should find the query terms in the lexicon, fetch the corresponding inverted lists from disk by performing a seek to the starting position of the list

inside the file, and traverse the compressed inverted lists in a DAAT fashion to return the ten highest-scoring results according to the BM25 ranking function. You should implement both conjunctive and disjunctive query processing. Each index posting should store a docID and either the frequency or a quantized impact score, but should not store positional information. After answering a query, your query processor should wait for the next input query, and for each query there should be a choice of disjunctive and conjunctive query mode.

Use the inverted index API framework provided in class to access inverted lists, thus hiding issues such as list decompression and file access and caching from the higher-level query processing algorithms. This means that you need to implement methods for opening and closing an inverted list, and for forward seeking in a compressed inverted list and retrieving frequencies or impact scores. You should not decompress inverted lists immediately after loading them from disk, but do the traversal on the still compressed lists, where local decompression is performed inside the seek operations on a lower level as needed.

It is recommended to compress docIDs and frequencies using varbyte encoding. If you use impact scores, a fixed 8-bit format for each score may be more appropriate. The docIDs should not be interleaved with frequencies or impact scores, but stored in separate blocks. You may also use other compression methods, such as Simple9, PEF, or many others, and you may reuse open-source code for compressing a posting or a single block – but not for doing the higher-level operations such as forward seeks. Do not use generic file compression techniques such as `gzip` or `bzip2` to compress inverted lists.

It is recommended to assign docIDs in the order in which the pages are parsed, and to not use any term IDs during indexing, but you can make your own choices. Also note that while the input data is mostly in English, it will also contain text from other languages that may use other character sets – at a minimum, your indexer should not fail in this case, and you may just ignore such text.

Your query processor may accept queries and return results using a simple command line interface. For extra credit, you may make it browser-accessible. For even more extra credit, you could return query-dependent snippets as part of the result page, using a suitable algorithm for selecting good snippets from the result passages. In this case, you would also have to store the indexed passages in a suitable format (say, indexed by offsets or in mysql), so that you can run your snippet algorithm on the top passages.

For this assignment, please hand in the following in electronic form: (a) your well-commented program source, and (b) a 10-15 page paper (written in Word or Latex or similar) explaining what your program can do, how to run the program, how it works internally, how long it takes on the provided data set and how large the resulting index files are, what limitations it has, and what the major functions and modules are. The paper should be readable to a generic computer scientist who knows what an inverted index is, but otherwise does not know much about search engines. So start very high-level! There will also be a demo for each group, scheduled close to the due date.