Project 3 – Search Engine

This assignment is to be done in groups of 1, 2 or 3, preferably the same groups that were in place for the crawler Project. Although this is presented as one single project that will take until the end of the quarter to complete, internally it is organized in 3 separate milestones, each with a specific deadline, deliverables and score. In doing milestones #1 and #2, make sure to look at the evaluation criteria not just of those milestones but also of milestone #3 –part of the milestones' evaluation will be delayed until the final meeting with the instructor.

You can use code that you or any classmate wrote for the previous projects. You cannot use code written for this project by non-group-member classmates. **You are allowed to use any languages and libraries you want for text processing, including the powerful nltk**. However, you are not allowed to use text indexing libraries such as Lucene, PyLucene, or ElasticSearch.

Goal: Implement a complete search engine. Your code needs to be able to perform well on the entire collection of crawled pages, under harsh operating conditions. This option is available to everyone, but groups that have at least one CS or SE student are required to do this.

Milestones Overview

	Goal	**Deliverables**	**Contribution for score**
#1	Produce an initial index for the corpus	Short report (no demo)	3
#2	Develop a retrieval component	Short report (no demo)	3
#3	Complete Search System	Code + Demonstration	54

General Specifications

You will develop two separate programs: an indexer and a search component.

1. Indexer:

Create an inverted index for the given corpus with data structures designed by you.
Tokens: all alphanumeric sequences in the dataset.
Stop words: do not use stopping, i.e. use all words, even the frequently occurring ones.
Stemming: use stemming for better textual matches. Suggestion: Porter stemming.
Important words: Words in bold, in headings (h1, h2, h3), and in titles should be treated as more important than the other words.

2. Search:

Your program should prompt the user for a query. This doesn't need to be a Web interface, it can be a console prompt. At the time of the query, your program will stem the query terms, look up your index, perform some calculations (see ranking below) and give out the ranked list of pages that are relevant for the query, with the most relevant on top. Pages should be identified by their URLs.

Ranking: at the very least, your ranking formula should include tf-idf scoring, and take the important words into consideration, but you should feel free to add additional components to this formula if you think they improve the retrieval.

Extra Credit:
Extra credit will be given for tasks that improve the quality of the retrieval and the of the search experience. For example:
• Detect and eliminate duplicate pages. (2 points)
• Implement Page Rank, and use it in your ranking formula. (3 points)
• Implement an additional 2-gram and/or 3-gram indexing and use it during retrieval. (2 points)
• Enhance the index with word positions and use that information for retrieval. (2 points)
• Index anchor words for the target pages (1 point).
• Implement a Web or GUI interface instead of a console one. (2 points)

Additional Specifications
Main challenges: design efficient data structures, devise efficient file access, balance memory usage and response time.
Corpus: all ICS web pages (developer.zip)
Index: Your index should be stored in one or more files in the file system **(no databases).**
Search interface:
The response to search queries should be less than 300ms. Ideally, close to 100ms, or less, but you won't be penalized if it's higher (as long as it's under 300ms).
Operational constraints:
Typically, the cloud servers/containers that run search engines don't have a lot of memory, but they need to handle large amounts of data. As such, you must design and implement your programs as if you are dealing with very large amounts of data, so large that you cannot hold the inverted index all in memory.

Your indexer must off load the inverted index hash map from main memory to a partial index on disk at least 3 times during index construction; those partial indexes should be merged in the end. Optionally, after or during merging, they can also be split into separate index files with term ranges. Similarly, your search component must not load the entire inverted index in main memory. Instead, it must read the postings from the index(es) files on disk**. I will check that both of these things are happening.**

Milestone #1

Goal: Build an index
Building the inverted index:
Now that you have been provided the files to index, you may build your inverted index off of them. The inverted index is simply a map with the token as a key and a list of its corresponding postings. A posting is the representation of the token's occurrence in a document. The posting typically (not limited to) contains the following info (you are encouraged to think of other attributes that you could add to the index):
• The document name/id the token was found in.
• Its tf-idf score for that document

Some tips:
• When designing your inverted index, you will think about the structure of your posting first.
• You would normally begin by implementing the code to calculate/fetch the elements which will constitute your posting.
• Modularize. Use scripts/classes that will perform a function or a set of closely related functions. This helps in keeping track of your progress, debugging, and also dividing work amongst teammates if you're in a group.
• I strongly recommend you use GitHub as a mechanism to work with your team members on this project.

Deliverables: Submit a report (pdf) to with the following content: a table with assorted numbers pertaining to your index. It should have, at least the number of documents, the number of [unique] tokens, and the total size (in KB) of your index on disk.

Evaluation criteria:
- Did your report show up on time?
- Are the reported numbers plausible?

Milestone #2
Goal: Develop a search and retrieval component
At least the following queries should be used to test your retrieval:
1 – Iftekhar ahmed
2 - machine learning
3 – ACM
4 – master of software engineering

Developing the Search component:
Once you have built the inverted index, you are ready to test document retrieval with queries. At the very least, the documents retrieved should be returned based on tf-idf scoring. This can be done using the cosine similarity method. Feel free to use a library to compute cosine similarity once you have the term frequencies and inverse document frequencies. You may add other weighting/scoring mechanisms to help refine the search results.

Deliverables: Submit a report (pdf) to with the following content:
• the top 5 URLs for each of the queries above
• a picture of your search interface in action
Evaluation criteria:
- Did your report show up on time?
- Are the reported URLs plausible?

Milestone #3
Goal: complete search engine
During this last stretch, you will improve your search engine in the following way. Come up with a set of at least 20 queries that guide you in evaluating how well your search engine performs, both in terms of ranking performance (effectiveness) and in terms of runtime performance (efficiency). At least half of those queries should be chosen because they do poorly on one or both criteria; the other half should do well. Then change your code to make it work better for the queries that perform poorly, while preserving the good performance of the other ones, and while being as general as possible.

Deliverables:
• Submit a zip file containing all the programs you wrote for this project, as well as a document with your test queries (no need to report the results). Comment on which ones started by doing poorly and explain what you did to make them perform better.
• A live demonstration of your search engine

Evaluation criteria:
- Does your search engine work as expected of search engines?
- How general are the heuristics that you employed to improve the retrieval?
- Is the search response time under the expected limit?
- Do you demonstrate in-depth knowledge of how your search engine works? Are you able to answer detailed questions pertaining to any aspect of its implementation?

Note: at the end of the project, you should have the optimized index that allows you to run both the indexer and the search with small memory footprint, smaller than the index size.

Understanding the Dataset

In Project 2, your crawlers crawled many web sites associated with ICS. We collected a big chunk of these pages and are providing them to you as a zip file: developer.zip. The developer.zip contains 88 domains found during crawling and a little under 56,000 pages.

The following is an explanation of how the data is organized.
Folders: There is one folder per domain. Each file inside a folder corresponds to one web page.
Files: The files are stored in JSON format, with 2 fields only:

- _"url" : contains the URL of the page. (ignore the fragment part, if you see it)
- _"content" : contains the content of the page, as found during crawling

Broken or missing HTML:
Real HTML pages found out there are full of bugs! Some of the pages in the dataset may not contain any HTML at all and, when they do, it may not be well formed. For example, there might be an open <strong> tag but the associated closing </strong> tag might be missing. While selecting the parser library for your project, please ensure that it can handle broken HTML.