Kick Covid in the _____.

Overview

The world is in a fight against the Corona Virus! What can computer scientists do in a situation like this? We can build tools to help epidemiologists, biologists, virologists, and other scientists search the incredible amount of data found in already-published research articles. A dataset consisting of thousands of scientific scholarly publications was created by the Allen Institute for AI in partnership with the Chan Zuckerberg Initiative, Georgetown University's Center for Security and Emerging Technology, Microsoft Research, and the National Library of Medicine - National Institutes of Health, in coordination with The White House Office of Science and Technology Policy.[1] The data set, named **CORD-19**, is a free resource to researchers containing currently 280,000 research papers.

For your final project in 2341, you're going to build a search engine for a subset of these articles that could help a scientist to efficiently find the information they are looking for.

Search Engine Architecture

Search engines are designed to allow users to quickly locate the information they want. Input to a search engine is a set of documents known as the **corpus**. Typically, the user will enter a search query, and any documents (the scholarly articles, in this case) that satisfy that query are returned to the user ordered by relevance. There is also the role of Search Engine Maintainer who directs the system in what documents to index, how to store the index and other maintenance tasks.

The four major components[2] of a typical search engine are the following:

1. Document parser/processor,
2. Query processor,
3. Search processor, and
4. Ranking processor.

Figure 1 provides a general overview of a search engine system architecture.

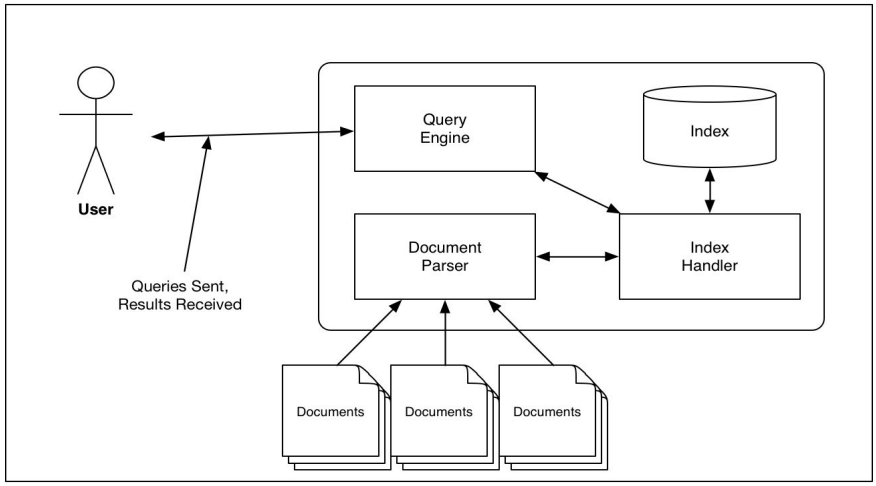

Figure 1 – Sample Search Engine System Architecture

[1] https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge/data
[2] http://www.infotoday.com/searcher/may01/liddy.htm

The fundamental "**document**" for this project is one scholarly research article with it's associated metadata such as authors and publication date. While we will be working with a subset of the articles, the full data set can be downloaded in its entirety from https://www.semanticscholar.org/cord19/download. The full dataset is currently 5.8GB zipped. It contains papers from multiple sources.

You can find the custom data set for our project at https://smu.box.com/s/w24j16y41d56jdwpl8wwq6r6d9r767ki.

The files are in JSON format. JSON is a "light weight data interchange format" (https://www.json.org/json-en.html). There are a number of open source JSON parsing libraries that are available for you to use in the project to allow you to quickly and efficiently extract the information from the files.

An explanation of the Parts of a Search Engine

The **index handler**, the workhorse of the search engine, is responsible for the following:

- *Reading from and writing to the main index.* You'll be creating an **inverted file index** which stores references from each element to be indexed to the corresponding document(s) in which those elements exist.
- *Searching the inverted file index based on a request from the query processor.*
- *Storing other data with each word.*

The **document parser/processor** is responsible for the following tasks:

- *Processing each research article in the corpus.* The dataset contains one file per article. Each document is in JSON format. Processing of an article involves the following steps:
    - *Removing stopwords from the articles.* Stopwords are common words that appear in text but that provide little discriminatory power with respect to the value of a document relative to a query because of the commonality of the words. Example stop words include "a", "the", and "if". One possible list of stop words to use for this project can be found at http://www.webconfs.com/stop-words.php. You may use other stop word lists you find online.
    - *Stemming words.* Stemming[3] refers to removing certain grammatical modifications to words. For instance, the stemmed version of "running" may be "run". For this project, you may make use of any previously implemented stemming algorithm that you can find online. One such algorithm is the Porter Stemming algorithm. More information as well as implementations can be found at http://tartarus.org/~martin/PorterStemmer/. Another option is http://www.oleandersolutions.com/stemming/stemming.html. C ++ implementation of Porter 2: https://bitbucket.org/smassung/porter2_stemmer/src.
- *Computing/maintaining information for relevancy ranking.* You'll have to design and implement some algorithm to determine how to rank the results that will be returned from the execution of a query. You can make use of metadata provided, important words in the articles (look up term-frequency/inverse document frequency metric), and/or a combination of several metrics.
- **Important Note:** Ignore any article that does not contain full text. You can find this information in metadata.readme file in the root folder of the archive.

The **query processor** is responsible for:

- *Parsing of queries entered by the user of the search engine.* For this project, you'll implement functionality to handle ***simple*** prefix Boolean queries entered by the user. The Boolean expression will be **prefixed** with a Boolean operator of either AND or OR i*f there is more than one word is of interest.* No query will contain both AND and OR. Single word queries do not need a boolean operator. Trailing

[3]See https://en.wikipedia.org/wiki/Stemming for more inform`ation.

search terms may be preceded with the NOT operator, which indicates articles containing that term should be removed from the resultset. A final optional operator of **AUTHOR** will allow the user to search for articles by a particular author's last name only.

- Here are some examples:
  - `virus`
    - This query should return all articles that contain the word *virus*.
  - `virus AUTHOR Donneley`
    - This query should return all articles that contain the word virus and in which one of the authors is Donneley
  - `AND biochemical virus`
    - This query should return all articles that contain the words biochemical ***and*** virus
  - `OR membrane barrier`
    - This query should return all articles that contain either membrane ***OR*** barrier
  - `AND membrane barrier NOT mitochondria`
    - This query should return all articles that contain membrane and barrier, but not mitochondria.
  - `membrane NOT mitochondria`
    - This query should return all articles that contain membrane, but not mitochondria.

- *Ranking the Results.* Relevancy ranking refers to organizing the results of a query so that "more relevant" documents are higher in the result set than less relevant documents. The difficulty here is determining what the concept of "more relevant" means. One way of calculating relevancy is by using a basic term frequency – inverse document frequency (tf/idf) statistic[4]. tf/idf is used to determine how important a particular word is to a document from the corpus. If a word appears frequently in document d_t but infrequently in other documents, then document d_t would be ranked higher than another document d_s in which a query term appears frequently, but it also appears frequently in other documents as well. There is quite a bit of other information that you can use to do relevancy ranking as well such as date of publication of the article, etc.

The **user interface** is responsible for:

- Receiving queries from the user
- Communicating with the Search Engine
- Formatting and displaying results in an organized, logical fashion

More info on the UI later.

The Index

The inverted file index[5] is a data structure that relates each unique word from the corpus to the document(s) in which it appears. It allows for efficient execution of a query to quickly determine in which documents a particular query term appears. For instance, let's assume we have the following documents with ascribed contents:

- d1 = `Computer network security`
- d2 = `network cryptography`
- d3 = `database security`

The inverted file index for these documents would contain, at a very minimum, the following:

- computer = d1
- network = d1, d2
- security = d1, d3
- cryptography = d2
- database = d3

[4]http://en.wikipedia.org/wiki/Tf-idf or http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html for more information
[5]See http://en.wikipedia.org/wiki/Inverted_index for more information.

The query "AND computer security" would find the *intersection* of the documents that contained *computer* and the documents that contained *security*.

- set of documents containing computer = d1
- set of documents containing security = d1, d3
- the intersection of the set of documents containing computer AND security = d1

Inverted File Index Implementation Details

The heart of this project is the **inverted file index**. You will implement this index with an AVL tree. Each node of the tree would represent a word being indexed and would provide information about all the articles that contain said word.

You will also keep an index of authors to process queries containing the **AUTHOR** operator. This index will also be an inverted index, but by authors' names rather than words in the articles.

Index Persistence

The index must also be persistent once it is created. This means that the contents of the index should be written to disk when requested by the user. The user should have the option of clearing the index and starting over.

User Interface

The user interface of the application should provide the following options:

- allows the user to clear the index completely
- allows the user to parse the corpus and populate index OR open a persistence file.
- allow the user to enter a properly formatted Boolean query (as described above).
    - The results should display the article's identifying/important information such as title, authors (at least the first), publication, date published. The result set shown to the user need not contain any more than the top 15 ranked articles. If you'd like to show more, you may wish to paginate.
    - The user should be allowed to choose one of the articles from the result set above and have the first ~300 words of the article printed.
    - Helpful Hint: that the query terms should have stop words removed and stemmed before querying the index.
- Print basic statistics of the search engine including:
    - Total number of individual articles indexed
    - Average number of words indexed per article (after removal of stop words)
    - Total number of unique words indexed (after stop word removal). Essentially, the number of nodes in the AVL Tree.
    - Total number of unique Authors.
    - Top 50 most frequent words (after removal of stop words)
    - Any other options you deem appropriate and useful.

Document Data Set

You can find the custom data set for our project at https://smu.box.com/s/w24j16y41d56jdwpl8wwq6r6d9r767ki.

Mechanics of Implementation

Some things to note:

- This project may be done individually or in teams of two students.
    - Individually: Finish all work on your own.
    - Team of 2 students:

        - Each team member must contribute to both the design AND implementation of the project.
        - Each class in the design must have an "owner". The owner is a group member that is principally responsible for its design, implementation and integration into the overall project.
    - This project must be implemented using an object-oriented design methodology.
- **You are free to use as much of the C++ standard library as you would like**. In fact, I encourage you to make generous use of it. You may use other libraries as well except for the caveat below.
    - You **must** implement your own version of an AVL tree and Hash Table (the storage data structures for the index). You may, of course, refer to other implementations for guidance, but you MAY NOT incorporate the total implementation from another source.
    - For the Hash Table, you can use a publically available hash function.
- You are free to use any library you can find to parse JSON.
- All of your code must be properly documented and formatted
- Each class should be separated into interface and implementation (.h and .cpp) files unless templated.
- Each file should have appropriate header comments to include the owner of the class and a history of updates/modifications to the class.

Submission Schedule

You must submit the following:

- **Teams:** Due Wednesday Nov 11 8, 2020 @ 10pm. Any name not submitted by this time is subject to being required to do the project individually. This info will be submitted via a Google Form to be distributed soon.
- **Initial Design Documents**: Due Saturday Nov 14 @ 6am uploaded to Canvas. *DO NOT think of this as your only task during week 1; this due date is to allow everyone to have one lab session before submitting.*
- **Milestone 1 and Parsing Speed Check:** Monday and Tuesday Nov 23 & 24th. More info on this coming soon.
- **Final Project: Due Friday Dec 4 @ 6:00am**
    - **No Late Submissions Accepted**
    - Complete project with full user interface
    - Documentation
        - UML Class Diagram
    - Demonstration of functionality to Professor Fontenot and TAs on Friday Dec 4 Via Zoom (sign up sheet to be distributed).

Thoughts and Suggestions

- If you wait even 1 week to start this project, you will likely not finish.
- A significant portion of your grade will come from your demonstration of the project to Prof. Fontenot and the TAs. Be ready for this.
- Take an hour to read about the various parts of the C++ STL, particularly the container classes. They can help you immensely in the project.
- As mentioned previously, beware of code that you find on the Internet. It isn't always as good as it seems. **Make sure that any code you use in the project is cited/referenced in the header comments of the project.**
- Take time to open a few of the articles in a text editor and examine them. Data is rarely beautiful and nicely formatted. However, this stuff is pretty good.

Grading:

- The final implementation project is worth *20% of your final grade* in this course (all other implementation projects are worth 35% percent of your final grade).
- Early Design Documents will count as a homework assignment.

- The Check In and Speed check will count as a homework assignment. The grade will not be based on speed, but only on completeness.
- The documentation for your project will count as a homework assignment