Did it Make the News?

Overview

Millions of people visit news websites daily. In some cases, a knowledge worker might need the ability to focus a search on more specific topics or industries. One very relevant and recent example related to Covid-19 is a dataset named Cord-19. A virologist or biochemist might be interested in targeted searches in the ~500,000 scientific articles it contains. Similarly, someone who works in the financial markets might be interested in a search engine that operates on financial new source data. And that is the type of data you're going to use in this project.

For your final project in CS 2341, you're going to build a search engine for a large collection of financial news articles from Jan - May 2018. The dataset contains more than 300,000 articles.

You can download the dataset from Kaggle at https://www.kaggle.com/jeet2016/us-financial-news-articles. You will need to make a Kaggle account to download it. Note that the download is around 1.3 GB and the uncompressed dataset is around 2.5 GB.

Search Engine Architecture

Search engines are designed to allow users to quickly locate the information they want. Building a custom search engine requires input of the documents that the user will eventually want to search. This is called the **corpus**. Then, once indexed, users can begin entering search queries. The search engine will take a query, find the documents that satisfy the request, and order them by some manner of relevancy. As you can see, there are two fundamental "roles" here: 1) the search engine maintainer, and 2) the search engine user.

The four major components[1] of a typical search engine are:

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.

The fundamental "**document**" for this project is one news article with its associated metadata such as publication venue, date, author, associated entities and the full text of the article.

The files containing the news articles are in JSON format. JSON is a "lightweight data interchange format" (https://www.json.org/json-en.html) that is easily understood by both humans and machines. There are a number of open source JSON parsing libraries available. The "officially supported parser" for this project is RapidJSON.

[1]http://www.infotoday.com/searcher/may01/liddy.htm

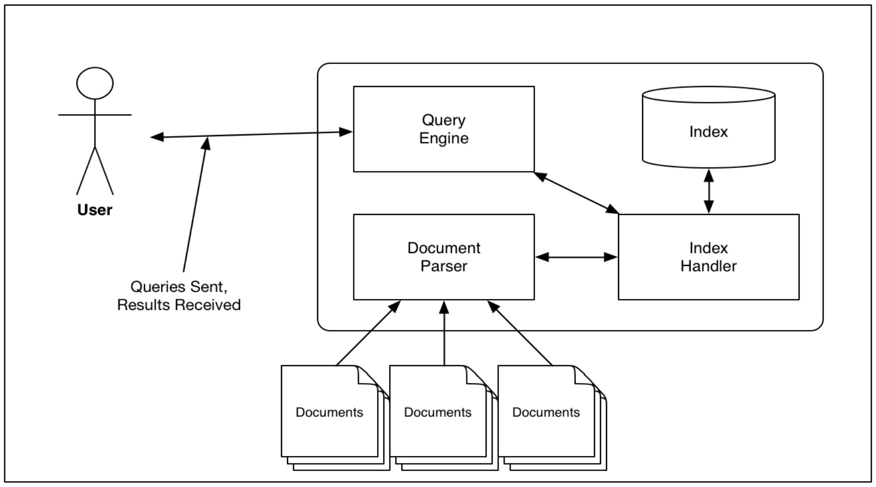

Figure 1 – Sample Search Engine System Architecture

An explanation of the Parts of a Search Engine

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

- *Read from and write to the main word 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.
- *Create and maintain an index of* **ORGANIZATION** *entities and an index of* **PERSON** *entities.*
- *Searching the inverted file index based on a request from the query processor.*
- *Storing other data with each indexed item (such as word frequency or entity frequency).*

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

- *Processing each news article in the corpus.* The dataset contains one news article per file. 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 useful information 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[2] 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

[2]See https://en.wikipedia.org/wiki/Stemming for more information.

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.

The **query processor** is responsible for:

- *Parsing 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 of interest*.
    - No query will contain both AND and OR.
    - Single word queries (not counting NOT or additional operators below) do not need a boolean operator.
    - Trailing search terms may be preceded with the NOT operator, which indicates articles containing that term should be removed from the resultset.
    - Additional Operators: A query can contain zero or more of the following:
        - ORG <some organization name> - the org operator will search a special index you maintain related to organizations mentioned in the entity metadata
        - PERSON <some person name> - the person operator will search a special index you maintain related to persons mentioned in the article's entity metadata.
        - Additional Operator Notes:
            - the order of ORG or PERSON doesn't matter (meaning, you should accept queries that have them in either order)
            - the operators will always be entered in all caps.
            - you may assume that neither ORG nor PERSON will be search terms themselves.
- Here are some examples:
    - `markets`
        - This query should return all articles that contain the word *markets*.
    - `AND social network`
        - This query should return all articles that contain the words "social" and "network" (doesn't have to be as a 2-word phrase)
    - `AND social network PERSON cramer`
        - This query should return all articles that contain the words social and network and that mention cramer as a person entity.
    - `AND social network ORG facebook PERSON cramer`
        - This query should return all articles that contain the words social and network, that have an entity organization of facebook and that mention cramer as a person entity.
    - `OR snap facebook`
        - This query should return all articles that contain either snap ***OR*** facebook
    - `OR facebook meta NOT profits`
        - This query should return all articles that contain facebook or meta but that do not contain the word profits.
    - `bankruptcy NOT facebook`
        - This query should return all articles that contain bankruptcy, but not facebook.
    - `OR facebook instagram NOT bankruptcy ORG snap PERSON cramer`

    - This query should return any article that contains the word facebook OR instagram but that does NOT contain the word bankruptcy, and the article should have an organization entity with Snap and a person entity of cramer

- *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[3]. 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 Index

The **inverted file index**[4] 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:

- doc d1 = `Computer network security`
- doc d2 = `network cryptography`
- doc 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

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**.

- To index the text of the articles, you will create an inverted 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.
- To index organizations and persons, you will use separate instances of an AVL tree.

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

In other words, you'll have 3 AVL Trees: one for the main index of words, one for organizations, and lastly, one for persons.

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 manage the persistent index (see Index Persistence above for more info)
- allows the user to parse a document dataset to populate the index OR read from the persistence file
- allow the user to enter a Boolean query (as described above).
    - You may assume the query is properly formatted.
    - The results should display the article's identifying/important information including Article Title, publication, and date published. If the result set contains more than 15 results, display the 15 with the highest relevancy. If less than 15 are returned, display all of them ordered by relevance. If you'd like to show more, please paginate.
    - The user should be allowed to choose one of the articles from the result set above and have the complete text of the article printed.
    - **Helpful Hint: that the query terms should have stop words removed and stemmed before querying the index.**
- Output basic statistics of the search engine including:
    - Total number of individual articles indexed
    - The total number of unique words indexed (total nodes in the word AVL Tree)
    - Any other interesting stats that you gather in the course of parsing

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 **<u>must</u>** implement your own version of an AVL tree. You may, of course, refer to other implementations for guidance, but you MAY NOT incorporate the total implementation from another source.
- You should research and use the RapidJSON parser. See https://rapidjson.org/ for more info. The other alternative is to create your own parser from scratch (which isn't as bad as it sounds).
    - RapidJson Tutorial made by TA Christian: https://github.com/Gouldilocks/rapidTutorial
- 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:** We will use a Github Classroom Group Assignment to create the repos. Make sure your Repo (either individual OR group) is created by 5pm on Friday April 8, 2022.
- **Week of April 11 in Lab**: Check in with TAs. You should have a complete AVL Implementation and be well on your way to parsing all the documents.
- **Parsing Speed Check:** Tuesday April 19, 2022. Check in and Parsing Timing Data Collection with Fontenot and TAs. Sign up sheet forthcoming.
- **Final Project: Due Tuesday May 3 @ 8:00am**
  - **No Late Submissions Accepted**
  - Complete project with full user interface
  - **Demonstration of functionality to Professor Fontenot and TAs** on Tuesday May 3 (Sign up sheet to be distributed later)

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

Opportunities for Extra Credit

Below are some additional features that you can implement to earn extra credit towards your PA01 - PA04 project grades. FYI, these additional features might require research on your part to complete.

Index Persistence

The index must also be persistent once it is created. This means

- the contents of the index should be written to disk when requested by the user,
- the contents of the persistent index should be read in when requested by the user and it should replace any data that is currently indexed in memory.
- reading the contents of the persistent index should be much faster than reparsing all the data from scratch.
- The user should have the option of clearing the persistent index and starting over.
- You can have a separate file for words, organization, and persons.

This feature is worth an additional 15 points.

Hash Table Index

You can implement a customized hash table class and use it to index the Orgs and Person entities (one hash table per). For the Hash Table implementation, you can use a publicly available hash function, std::hash(), or implement one of your own.

This feature is worth an additional 15 points.

Gather Additional Statistics

- When processing a query from the user, use std::chrono to time how long it takes to execute the complete query and output this at the top of the query results.
- Total number of unique Orgs and Person entities.
- Top 25 most frequent words in descending order (NOT including stopwords)

This feature is worth an additional 10 points.