Overview

Millions of people visit news websites daily. 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. 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 measure of relevancy.

Figure 1 provides a general overview of a search engine system architecture. The main parts are the document parser which extracts keywords to be indexed by the index handler, and the query engine which uses the index to answer user queries. The "**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** (https://rapidjson.org/). The code is already included with an example in your project template.

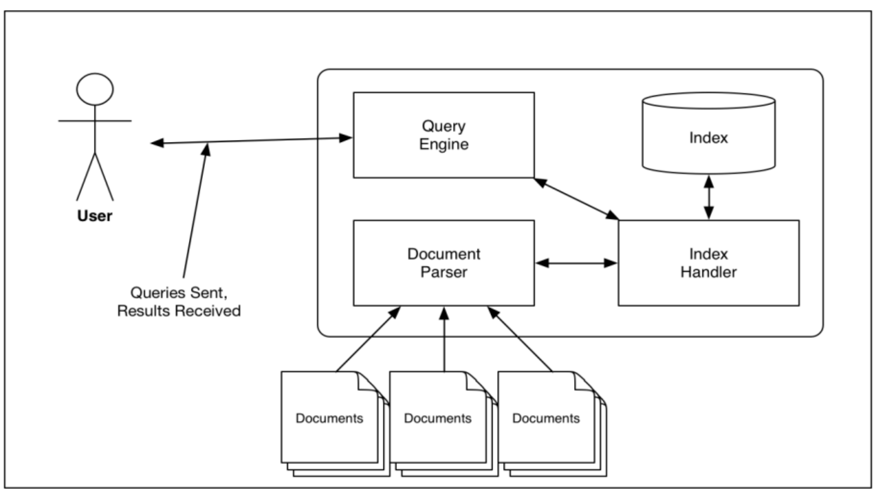

Figure 1 – Sample Search Engine System Architecture

An Explanation of the Parts of a Search Engine

The **index handler**, the workhorse of the search engine, is responsible for reading and writing to the index. An index is an **inverted file index** which stores references from each index term to the corresponding document(s) in which those terms exist. The index may also store additional information like term frequencies. It also searches the inverted file index based on a request from the query processor.

You will create three indices:

1. Create and maintain the main index for terms (words) in the documents.
2. Create and maintain an additional index for **ORGANIZATION** entities
3. Create and maintain an additional index an index of **PERSON** entities.

The **document parser** 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 (https://en.wikipedia.org/wiki/Stemming) 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.

The **query processor** is responsible for:

Parsing queries entered by the user of the search engine. For this project, you will start with a simple interface like google’s search interface where all entered terms need to be in the result. This is equivalent to a logical AND. Terms that are preceded by a `-` cannot be in the document.

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*.
- **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)
- **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.
- **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.
- **facebook meta -profits**
  - This query should return all articles that contain facebook and meta but that do not contain the word profits.

Note on ORGs and Person names that consist of multiple words. You can implement them in the following way:

- **social network PERSON:Mark PERSON:Zuckerberg**

A better query processor that supports AND and OR could be implemented using a parse tree, which is optional.

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. 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** (see http://en.wikipedia.org/wiki/Inverted_index) is a data structure that relates each unique term 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 “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

In addition, you probably need to efficiently match document IDs like d1 to actual file names.

Inverted File Index Implementation Details

The heart of this project is the **inverted file index**. The goal is to efficiently find terms in the indices.

- 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 using an appropriate data structure.
- To index organizations and persons, you will use separate instances of an AVL tree.
- Your implementation of AVL Tree will be a version that has a key (the term) and values (the documents) and will operate like a map (see https://en.cppreference.com/w/cpp/container/map). Note that your values can be a list of documents.

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

Persistence

Often, we need to keep the information stored in data structures between two runs of a program. You need to implement a way to store each AVL Tree as a file and read it back into memory again. This is called persistence since the data structure persists over multiple runs of the program. Typical solutions include Google protocol buffers https://developers.google.com/protocol-buffers/docs/cpptutorial and similar libraries. Converting objects or data structures into a sequence of bytes that can be stores or transmitted is called serialization (https://en.wikipedia.org/wiki/Serialization). Popular formats include XML, JSON, YAML, etc.

For this exercise you have to **implement your own persistence mechanism.** You need to write a member function that saves the information in your AVL tree into a text file and then a second function (a constructor) that reads the text file and recreates the AVL tree in memory. You will need to think about an appropriate structure for this file so it is easy to create, read and convert back into a tree. A simple approach would be to create a text file and write each tree node as a line with different information separated by semicolons (this is similar to CSV).

User Interface

Command line interface: The program needs to be called `supersearch`. The first argument is the command followed by additional information for the command. Here are examples:

Usage:

Index all files in <directory> and store the index in one or several files:
supersearch index <directory>

Load the existing index and perform the following query:
supersearch query "social network PERSON:cramer"

Start a simple text-based user interface that lets the user create an index,
save/load the index and perform multiple queries:
supersearch ui

For the UI, you can display a simple menu like "Press i for creating a new index," etc. It should provide the following options:

- allows the user to create an index from a directory with documents.
- allows the user to write the index to a file (make it persistent) and read an index from a file.
- allow the user to enter a query (as described above).

Some general points:

- 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.

Design Requirements:

- This project must be implemented using an object-oriented design methodology.
- **Use an AVL tree and persistence mechanism for the main word index.**
- **Use the C++ standard library.** Document and explain all code/libraries that you use.
- Use the RapidJSON parser. See https://rapidjson.org/ for more info. You may use other parsers, or create your own.
- Code must be extensively documented and well 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.

NOTE:

- Create a design. Make your UML and program structure.
- A significant portion of your grade will come from your demonstration of the project.
- Use the C++ STL, particularly the container classes. They can help with the project.
- **Any code you use in the project must be cited/referenced in the header comments of the project.**