Document Search Application

Written By **- Akshay Agarwal**

Programming Language used - Python 3.7

Goal – This project performs document search and count the occurrences of the input search term in 3 input data files.

Application Inputs & Output:

Inputs – The execution of main module takes following 2 inputs from the user:

1. Search term – This is the search term which user is trying to count the occurrence of search term in the documents.

2. Search method – This is the search method which user wants to use, to count the occurrence of search term in the documents. Allowed input values for search methods are:
    - simple_search
    - regex_search
    - index_search

Output – The code will return the count of occurrences in the 3 text files in the descending order of relevance. Higher relevance means higher count of occurrences of the search term in that document. An example output of search term *of* using index_search method is as follows:

Number of matches of word 'of' in french_armed_forces.txt - 27

Number of matches of word 'of' in hitchhikers.txt - 13

Number of matches of word 'of' in warp_drive.txt - 5

Project Structure

The project is organized in the following structure:

main – This is the package consisting of modules for all three types of search methods of this application. This package also consists of utils and prep_benchmark packages.

utils – This package consists of the common utility modules used across the project.

prep_benchmark – This package consists of the modules used for preprocessing and benchmarking.

test – This package includes the test cases of this application.

data – This directory consists of the source data files.

docs – This directory consists of document describing the project details along with all the execution steps for the user of this application.

Steps to execute the Application:

Prerequisite – Please ensure Python 3.7 is installed on the system from where the code is being executed and python installation is added to the PATH environment variable, to execute application's executable archive.

Execution Steps:

1. Clone the Git Repository from using git clone into the desired local path.

2. Navigate to the cloned project directory:
   $ cd document_search

3. Execute the code using the following command:
   python document_search_application.zip
   a) Upon execution the user would be prompted with the following message, to enter the search term:
   ```
   Please enter the search term:
   ```

   Example input value:-
   ```
   Please enter the search term: of
   ```

   Here please enter the search term which you would like to search in the data files.
   Press Enter key(on Windows OS) or return key (on Mac OS).

   b) Next user would be prompted with the following message, to enter the type of search method:
   ```
   Please enter the search method:
   ```

   Example input value:-
   ```
   Please enter the search method: index_search
   ```

   Allowed input values for search method:
   - simple_search
   - regex_search
   - index_search

Press Enter key(on Windows OS) or return key (on Mac OS).

4. Output – The code will print the count of occurrences of search term, in the input data files in descending order of relevance. Higher relevance means higher count of occurrences of the search term in that data file. Example output of search term *"of"* using index_search method is as follows:

```
Number of matches of word 'of' in french_armed_forces.txt - 27
Number of matches of word 'of' in hitchhikers.txt - 13
Number of matches of word 'of' in warp_drive.txt - 5
```

5. The location of the source data files is controlled by config.json file in conf directory. The default populated path in the config file points to the source files present inside project's data directory. If required this path can be configured to a different path where the data files are located. **Note** - For paths on Window operating system please specify the '\' escape character for escaping \ in the path value. For example - a path for Window OS, having the data files residing inside E drive at E:\data_files\src_path\ should be as follows:

```
{
  "src_files_base_path": {
    "path": "E:\\data_files\\src_path\\"
  }
}
```

Performance Benchmarking:

Time elapsed for 2 Million random words:

elapsed time for simple search = 113.16435313224792 sec

elapsed time for regex search = 228.7137370109558 sec

elapsed time for index search = 4.6585307121276855 sec

This benchmarking **calculates the time elapsed** by each search method, **to return the dictionary with the files name as key and counts of searched word in that file as value, in descending order of count values**, when sequentially executed for random 2 Million words.

Cause of index search being most efficient - Here the index search executes fastest because it searches in a preprocessed JSON file, consisting of counts of words in each of the 3 data files stored as key value pairs. The JSON file I have created consists of the files names as key and a dictionary as value. The value dictionary consists of the word in that file as key and the count of that word as value. One subset of the records of preprocessed JSON looks like follows:

```
{"french_armed_forces.txt": { "The": 7,……
```

Here the key is the name of file french_armed_forces.txt and the value for this key, which is also a dictionary has the word 'The' as Key and value 7. Here value 7 denotes the number of occurrence of word 'The' in file french_armed_forces.txt

Now **the index search is fastest because** the **time complexity of search operation in a dictionary is constant O(1)**. Hence this does not depend on the size of the file or the number of words in the file which is the case with the other simple and regex search operations. Hence there is this drastic difference in performance.

Changes at hardware and software level to handle large number of requests-

Considering actual volume of data would be much larger than the given sample files and to handle large number of requests per second, we should choose a distributed NoSQL system such as Cassandra. Cassandra is efficient for read operations and can efficiently handle high throughput. Since we know ahead of time what the user requests would look like, which are around the count of search terms in the document. Hence we can load the preprocessed JSON data into Cassandra and utilize its partitioning strategy to partition the data for efficient read results. The other way of preprocessing to produce json(dictionary) data can be, one with the search terms(knowing the splitting logic of the data in file) as key and the value as a dictionary. The value which is a dictionary should have the file name as key and the number of occurrence as value. We can sort the dictionary(which is a value of main dictionary) by value. Thus we can directly fetch the result based on the key(which is search term). This would make the execution of search requests, for a particular search term in the files highly efficient. Also the distributed architecture would offer fault tolerance and no single point of failure(no master) architecture would ensure high availability.