Programming Assignment 1 - K most popular words

Hui-Yu Liu, Ching-Yueh Huang

I. Introduction

Nowadays, information from large datasets has become a fundamental task in many domains. In this context, identifying the most frequent words in a text corpus is a common problem that has many applications, from information retrieval and natural language processing to social media analysis and marketing. In this assignment we are going to find the top k frequent word in the big dataset, which is 300MB, 2.5GB and 16GB. We present an analysis of the performance of our implementation of a top-K most frequent words algorithm using different input dataset sizes and optimization techniques. We evaluate the algorithm's efficiency in terms of execution time, memory usage, and CPU utilization, and we analyze the impact of different algorithms, data structures, and system resources on its performance.

II. Methodology

Counting repeated objects is a fundamental problem in programming, particularly when dealing with large datasets. One common application of this problem is finding the top K words in a file. Fortunately, Python provides a range of tools to help solve this problem. For instance, the Counter class is an integer variable with an initial value of zero that we can increment to reflect the number of times a given object appears in an input dataset. With the Counter, we can easily find the top K words by using the most_common(k) method.

However, when dealing with a large dataset, the Counter class may not be sufficient. While it may work for a 50MB or even a 300MB file, it may take an excessively long time to count the words in a 2.5GB or 16GB file, something like “forever”. To overcome this challenge, we decided to shard our large dataset into smaller files. Sharding involves splitting the dataset into multiple chunks, with each chunk being a manageable size for counting the words.

To shard our dataset, we first obtained the file size using the command "wc -c < {}" (where {} is the file path) and stored the result in the variable "file_size".

```
file_size = int(os.popen('wc -c < {}'.format(file_path)).read())
```

We then split the file into multiple chunks by setting the chunk size to be the file size divided by the desired number of shards, rounded up using the numpy ceil function. With smaller size files, we can easily create frequency dataframes using Pandas, where we set two columns for "word" and "count". Before applying the words into the dataframe, we converted all the input into lowercase and checked that only alphabetic words were considered. Once we finished creating the dataframes, we sorted the values by "count" and found the top K words.

```
    # Convert word frequencies to Pandas DataFrame and sort
    freq_df = pd.DataFrame({'word': list(freq.keys()), 'count':
list(freq.values())})
    # Only keep alphabetic words
    freq_df = freq_df[freq_df['word'].apply(lambda x: x.isalpha())]
    freq_df = freq_df.sort_values(by='count',
ascending=False).reset_index(drop=True)

    # Print top k words and their frequencies
    print(freq_df.head(k))
```

To accurately count and find the most frequent words in a large dataset, sharding it into smaller files and utilizing Pandas dataframes can help overcome various challenges. By converting inputs into lowercase and filtering out non-alphabetic words, we can precisely count and identify the most common words in the dataset. It's important to note that a set of stopwords is provided, which is only in lowercase, and must be skipped to ensure accurate word counts.

When processing such large files, using a multiprocessing pool can significantly speed up the computation time. After we break the file into smaller shard files, we can utilize a pool of worker processes to process each shard in parallel. Once the shard size has been determined, we can create a multiprocessing pool and pass the shard files as arguments to the worker function that performs the word count. The pool automatically distributes the workload across the available worker processes, allowing us to process multiple shards in parallel and speed up the computation time.

```
    # Set up multiprocessing pool
    pool = mp.Pool(num_processes)
```

```
    with open(file_path) as f:
        for i in range(shard):
            chunk = f.read(chunk_size).lower()
            # apply_async() schedules the function to be executed
asynchronously
            # in a separate process and returns a 'asyncresult' object
            # which is added to the results list
            results.append(pool.apply_async(count_words,
args=(re.findall(r'\w+', chunk), stop_words)))
```

III. Result and Analysis

At first, we are trying to use chunk size to generalize the file into different numbers of sharding files. But we found it is not intuitive while testing data. So we change the input variable to shard to make it easier to understand.

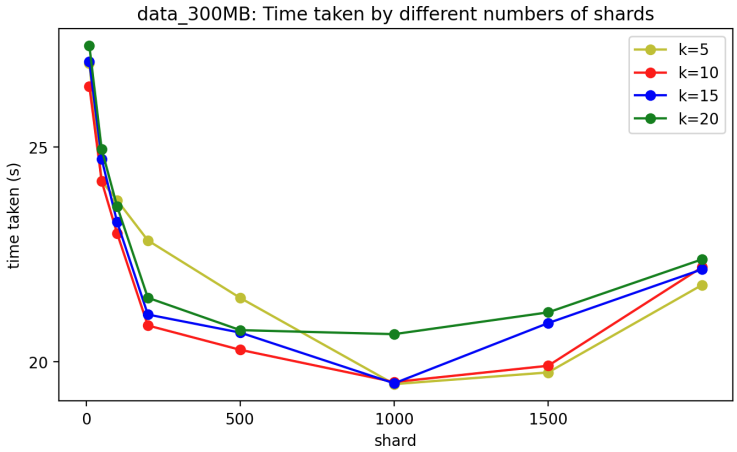
```
top_k_words(file_path, stopword_path, k, shard, num_processes)
    """
        Input: file_path - path to file
               stopword_path - path to stopword file
               k - number of top words to return
               shard - number of shards to split file into
               num_processes - number of processes to use
        Output: DataFrame with top k words and their frequencies
                sorted in descending order
    """
```

For the num_processes, the default number will be the number of logical CPU cores in your systems. Take us as an example, we are using Macbook Pro, with 4 cores and 16GB memory. So we will set the num_processes to 4 as default. The hardware overview is below:

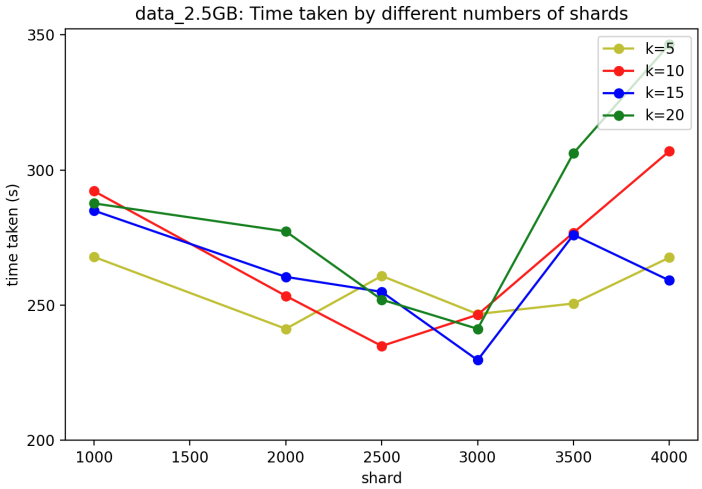
Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro16,2
Processor Name:	Quad-Core Intel Core i5
Processor Speed:	2 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	512 KB
L3 Cache:	6 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB
System Firmware Version:	1968.100.17.0.0 (iBridge: 20.16.4252.0.0,0)

We conducted performance testing on two datasets, data_300MB.txt and data_2.5GB.txt, to evaluate the efficiency of our top-k frequent word algorithm. As illustrated in the graph, the execution time varies with different values of k and shard numbers. The graph clearly demonstrates the effect of varying the number of shards on the algorithm's performance, with a reduction in execution time observed as the number of shards increases. By analyzing the graph, we can determine the optimal configuration for the algorithm given the size of the dataset and the desired value of k.



Our results show that the execution time of the algorithm is strongly influenced by the number of fragments. As the number of shards increases, the execution time of the algorithm decreases significantly. Specifically, for data_2.5GB.txt, with k=10, we observe a reduction in execution time of about 16% when using 3000 shards compared to using 1000 shards. Similarly, for data_300MB.txt, we observe that the execution time is reduced by approximately 27% when using 1000 shards compared to using 10 shards.



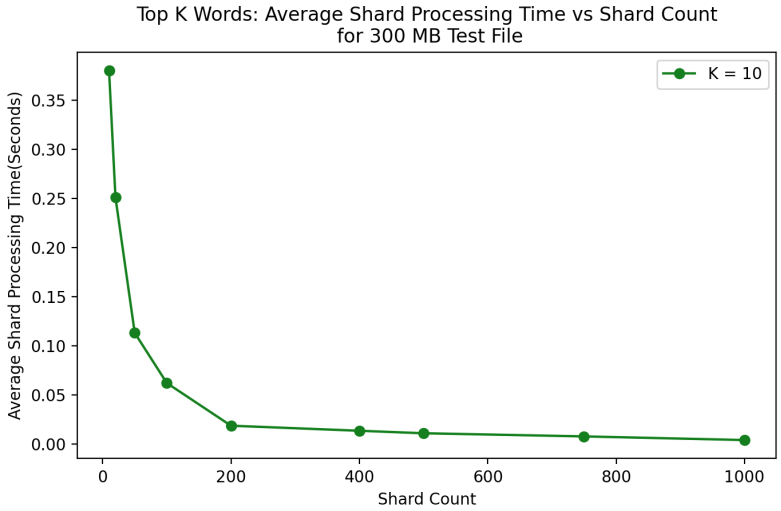
Moreover, our results show that the value of k also has an impact on the execution time of the algorithm. As k increases, the execution time also increases. However, the effect of k on the execution time is not significant compared to the effect of the number of shards.Based on our analysis, we recommend using more shards when processing large datasets to improve the efficiency of the algorithm. The optimal number of shards depends on the size of the dataset, and we found that for data_2.5GB.txt, using 2500 - 3000 shards provided the best performance. For data_300MB.txt it is about 1000 shards. Users can consider the value of k when choosing the number of shards to use in order to strike a balance between efficiency and accuracy.

We utilized various scales and metrics to evaluate our performance on the data_2.5GB file, initially focusing on the time taken for shard sizes ranging from 250 to 2000 bytes. Our observations indicated that increasing the shard size improved performance. Subsequently, we scaled up our tests with larger shard sizes ranging from 1000 to 4000, ultimately determining that the optimal performance was achieved with around 1500 shards.

We also make an analysis of the average shard processing time vs the shard count, take K=10 as an example. This provides an insight into algorithms on how sharding involves the process. The average shard processing time decreases as the number of shards increases, can provide insights into the performance of the algorithm and help determine the optimal configuration for the algorithm given the size of the dataset and the desired value of k.

Throughout our testing phase, we discovered that the shard count must be carefully configured. The shard count determines the size of each individual shard file. To achieve

efficient file processing and accurate word count, it's essential to set the shard value such that each shard file is between 5-10 MB in size. When we execute the program using a multiprocessing pool with this shard size, it performs correctly and efficiently.



IV. Conclusion

In this project, our goal was to create a program that could analyze the correlation between input size, algorithm, data structure, and system resource utilization. We accomplished this by using Matplotlib Pyplot to create several visualizations that helped us understand how the shard file size and number of chunks affected our performance. To optimize our program, we implemented a multiprocessing pool with an asynchronous mechanism, and used Pandas Dataframe to store the frequency values that met our requirements. Additionally, we determined the appropriate chunk size based on the file size, and sharded the file into chunks using the techniques we adopted. By utilizing these techniques, we were able to successfully scale our program to handle larger datasets.

V. Reference

What is Sharding? - GeeksforGeeks

Python's Counter: The Pythonic Way to Count Objects – Real Python

pandas.Series — pandas 2.0.0 documentation

multiprocessing — Process-based parallelism — Python 3.11.3 documentation