# **K Most Popular Words**

**Introduction**

The objective of this project is to find the Top K most frequent words in a given dataset while minimizing the execution time. The implemented code should be able to handle different input data sizes while providing optimal performance. The code is designed to work in a parallelized manner using multiprocessing, aiming to take advantage of multiple CPU cores to improve performance. The code also avoids analyzing common "stop" words to reduce the amount of unnecessary processing.

**Basic approach:**

***Data structure used:*** Hashmap and List

The input file is read using a buffer reader, the frequency of each word is stored in a hashmap.The map entries are added to a list and sorted using a custom comparator that sorts by word count and lexicographical order. When printing the top K words, the program compares them to a stop word which is stored in a hashset resulting in constant search time on average, compared to binary search, which takes O(logN) time per search.

**Version 2**

***Data structure used:*** Hashmap and Priority queue(Min-Heap)

Instead of sorting all unique words in a list, a ***min-heap with a custom comparator*** is used. The algorithm ensures that the heap only holds K frequent elements at any given time, with a space complexity of O(K).New elements are inserted by checking against just the root element and are inserted in O(logK) time.This approach outperforms using a max-heap, which would require O(N) space and O(logN) insertion time.

Also to minimize disk access and optimize execution time, the file is read in chunks of variable size instead of processing it line by line.

**Version 3**

***Data structure used:*** Hashmap and Priority queue(Min-Heap) with threadpool

To fully utilize CPU resources, a multithreading approach was adopted using a thread pool consisting of threads equivalent to the number of cores.. Each thread reads a data chunk from the file using ***RandomAccess library*** Algorithm of version 2 is used for processing words. Each thread has a local concurrent hashmap that stores the count for its respective data chunk. Once each thread completes computation, the local hashmaps(N) are merged with the ***global concurrent hashmap***(M). While parallel processing reduces execution time for large datasets, merging the hashmaps incurs an additional overhead of O(NlogM).

**Final version**

**ALGORITHM**

**Step 1 -** Download the list of stop words**.**

**Step 2 -** Read the dataset file into memory.

**Step 3 -** Divide the content into chunks for parallel processing.

**Step 4 -** Process each chunk in parallel, filtering out stop words and counting the frequency of the remaining words.

**Step 5 -** Merge the results from parallel processing.

**Step 6 -** Extract the top K most frequent words from the merged results.

**Step 7 -** Analyze and print performance metrics.

**DATA STRUCTURES USED:**

1. **Set - For Stop Words :** Set is used to store stop words, providing efficient O(1) time complexity for checking the presence of a word. This allows for fast filtering of stop words from the dataset.
2. **List - For Chunks :** The dataset content is divided into chunks, represented as a list of strings. This allows for easy distribution of work among multiple processes.
3. **Counter :** The collections.Counter class is used to count word frequencies efficiently. Counters can be easily merged, which is beneficial when combining the results from parallel processing.

**DESIGN**

1. **MULTIPROCESSING :** In our code we are using the “multiprocessing.Pool” to parallelize the processing of dataset chunks. This is allowing the program to take advantage of multiple CPU cores, significantly reducing the execution time for large datasets.
2. **MEMORY-MAPPED FILE (MMAP) :** In our code we use “mmap” to read the dataset file, which allows the program to handle large files efficiently without loading the entire file into memory at once.
3. **REGULAR EXPRESSIONS (re) :** We have used the “re” module to remove characters other than alphabets and hyphens from the dataset chunks. This ensures that the processed words only contain valid characters.
4. **PSUTIL :** The “psutil” module is used to collect performance metrics, such as CPU utilization and memory usage, providing valuable insights into the program's efficiency.

**TIME COMPLEXITY**

The total time complexity of our code can be represented as O(N/P) + O(P*log(P)), where N is the number of words in the dataset and P is the number of processes. The time complexity is divided into two parts: parallel processing of chunks and merging the results. The parallel processing of chunks has a time complexity of O(N/P), while the merging of results from P processes has a complexity of O(P*log(P))

**System Specification**

Model Name: MacBook Pro

Chip: Apple M1 Pro

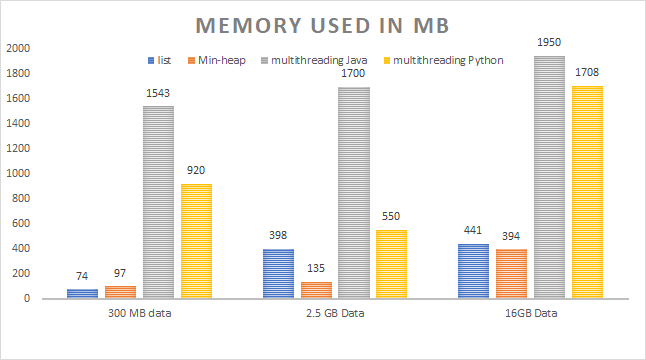
Total Number of Cores: 8 (6 performance and 2 efficiency)

Memory: 16 GB

**Comparison of Performance metrics**

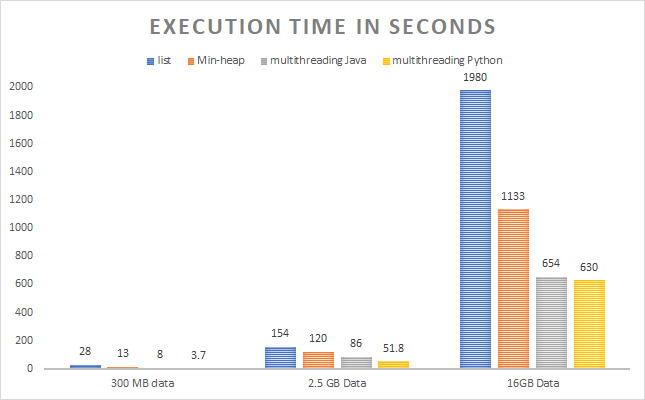
***1.Memory Usage***

The memory usage of list and heap implementation is lower as they ready the file sequentially has only one chunk of data in the memory at a time.The heap method O(K) uses comparatively lesser memory to the list implementation O(N) as its hold only K elements.In the Multithreading approach each thread will have a local hashmap,increasing the memory usage, and it involves creating multiple processes which process multiple chunks in parallel,each of which requires its own memory space for execution.



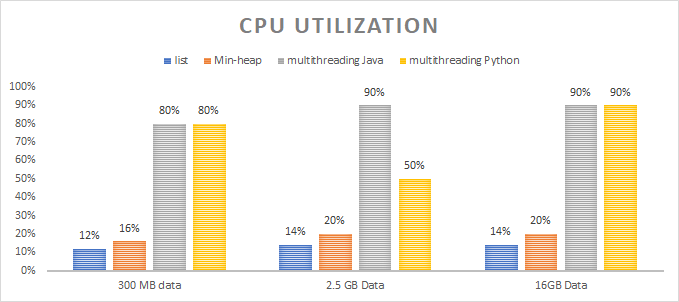
***2.Execution time***

The list uses O(NlogN) , the Heap has O(NlogK) time complexity,however as the dataset size increases, the sequential processing of the entire dataset can become a bottleneck, resulting in slower execution times.

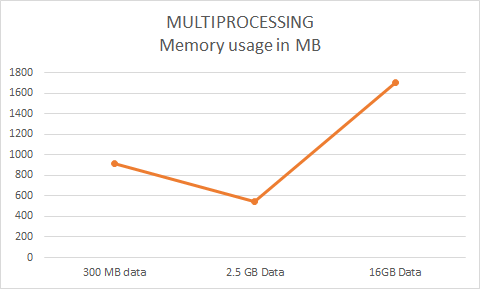


***3.CPU Utilization***

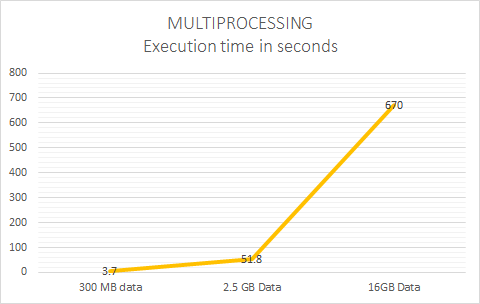
In sequential processing using list and heap only one thread is used to read and process the dataset, which can result in lower CPU utilization but increased execution time even if the dataset increases.



***Memory Usage:***

* ****
* As the Memory usage increases because more data needs to be processed and stored.
* The non-linear increase in memory usage can be attributed to the efficient handling of large files using memory-mapped files (mmap), which allows the program to read and process the data without loading the entire file into memory.

**Execution Time**



* **Parallel processing allows the code to process larger datasets more efficiently by distributing the workload among multiple CPU cores.**
* **The execution time still increases with dataset size because more data needs to be processed, but the increase is mitigated by the parallel processing.**