K Most Popular Words

Assignment-1 Report

Ben Chen
Joy Huang

Introduction

The goal of this assignment is to optimize the performance of reading a large-size text file, to do simple cleaning, to count the frequency of the words and to return the top k words with highest frequency.

To optimize the performance of reading a large-size text file, cleaning the text, and counting the frequency of words, we divided the task into several steps:

STEP 1: **Reading the text file efficiently**

- Use buffered reading techniques to minimize I/O operations.
- Read the file in chunks or lines rather than loading the entire file into memory at once.

STEP 2: **Cleaning the text**

- Convert the text to lowercase to ensure case-insensitive word counting.
- Split the text into individual words or tokens.

STEP 3: **Counting the frequency of words**

- Use a data structure like dict, defaultdict or self-implemented trie to store the word-frequency pairs efficiently.
- Iterate through each word in the text and update the corresponding frequency count in the data structure.
- Skip any stop words are not essential for word frequency analysis.

STEP 4: **Returning the top k words with the highest frequency**

- Sort the word-frequency pairs based on the frequency in descending order.
- Select the top k words from the sorted list.
- If k is relatively small compared to the total number of unique words, it is better to use a min-heap data structure to efficiently keep track of the top k words as we iterate through the text.

Methodology

1. Environment setup:

Device: MacBook Pro M1 14" 2021, CPU 8 cores, 16GB RAM
Language: Python 3.11 / (Cython)

2. Data Source:

Three text files(.txt file) with different sizes: 50MB, 300MB, 2.5GB, 16GB

- Test:
  https://drive.google.com/file/d/1mJTQbWJp-n5VAEx0dk5PeeSLo67XfbjG/view?usp=sharing
- Regular:
  https://drive.google.com/file/d/1mJTQbWJp-n5VAEx0dk5PeeSLo67XfbjG/view?usp=sharing

3. Data Structure Algorithm Design
   We tried different methods to speed up the baseline, we started from observing the execution time breakdown from the baseline code. Even though we find that the breakdown varies between different input sizes, we can still get insight about how we optimize the part with the greatest impact.

   a. **Baseline**
      i. Sorting algorithm: Heap-sort
      ii. Reading practice: Read all text in RAM upfront.
      iii. Text cleaning: default `split` and `lower` function
      iv. Counting: Defaultdict with hashmap for stop words.

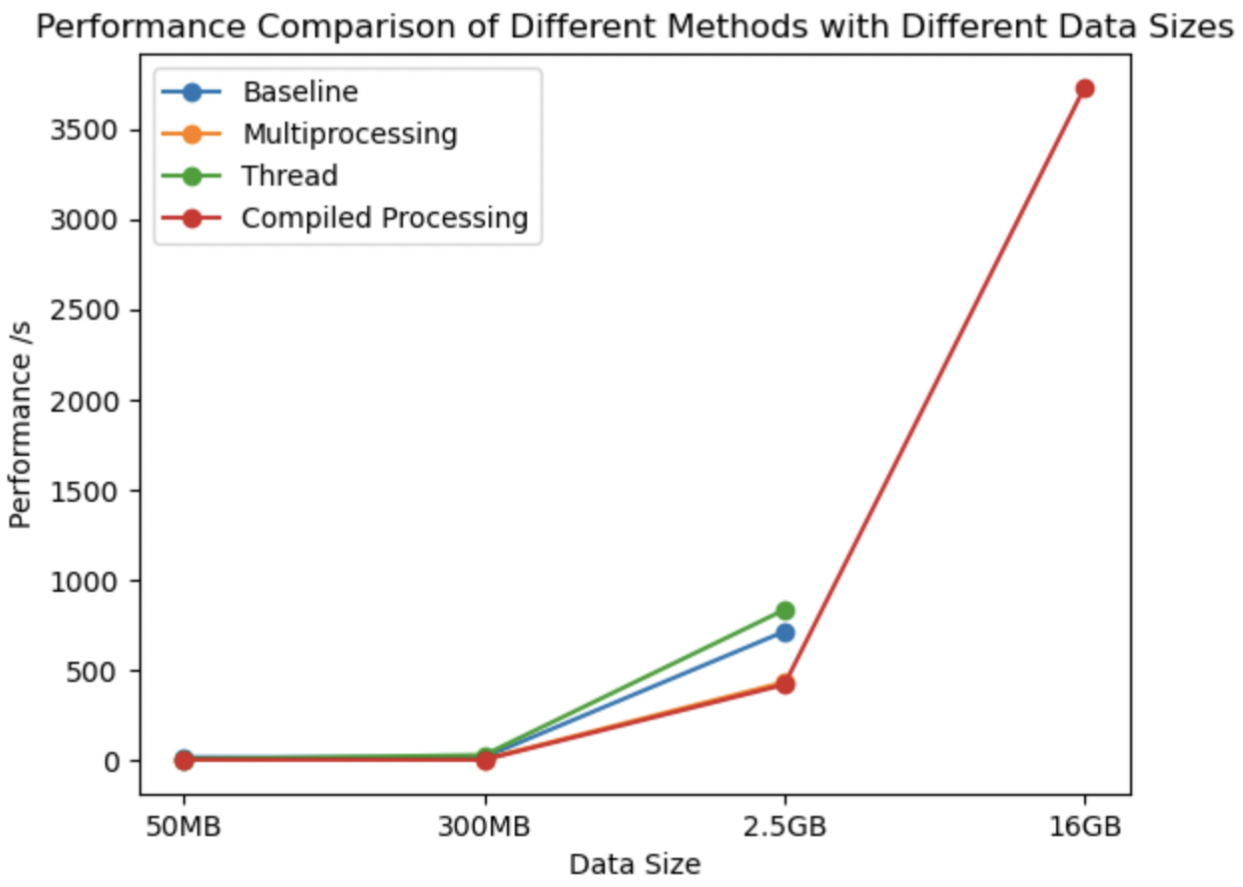
   b. **Choice of the Counter:**
      i. collection.Counter object is significantly slow then plain dict.
      ii. Trie is good for memory saving, but in terms of a single word look up, it requires multiple hash functions to be calculated, therefore it's not suitable for our task.

   c. **Multiprocessing**
      i. For parallel map reduce strategy to speed up the code, multithreading wont work because of two things. The first is GIL will keep python from truly doing parallel computing; the second is the job is not I/O bound, therefore, the switching thread will only add up overhead here. So we only put multiprocess in the report.
      ii. For multi-processing, we sacrifice some expensive inter-process communication but enjoy the significant performance improvement on the word counting part by applying map-reduce strategy to count different partitions on different CPU cores, and eventually aggregate them to get the final answer.

   d. **Compiled python**

i. Another optimization for the computation heavy task, we compile the most computation intensive part of the code, using a strong-typed Cython programming language.
ii. We keep the multi processing part of the code and substitute the word_count and text processing part by compiled Cython implementation.

Performance



It is obviously seen from the graph that among 4 methods we tried, Compiled Processing has the highest efficiency through all the data sizes we proceeded with.

When the data size is small, the performance differences between various data systems may not be significant. However, as the data size grows, it becomes more common to observe a surging trend in the runtime cost of data systems.

This trend can be attributed to several factors, for example:

1. **Increased computational complexity**

With larger data sizes, the computational complexity of processing and analyzing the data tends to increase.

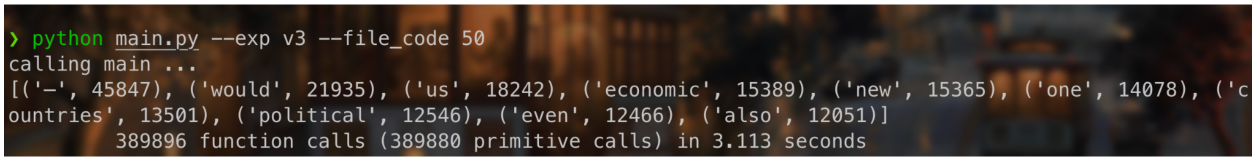
2. **Memory limitations**
   When data cannot fit entirely in memory, systems may resort to disk-based storage, which can significantly impact performance
3. **Scalability challenges**
   Bottlenecks in distributed systems, suboptimal parallelization, or limitations in hardware resources can hinder the system's ability to handle larger datasets effectively.

Limitations

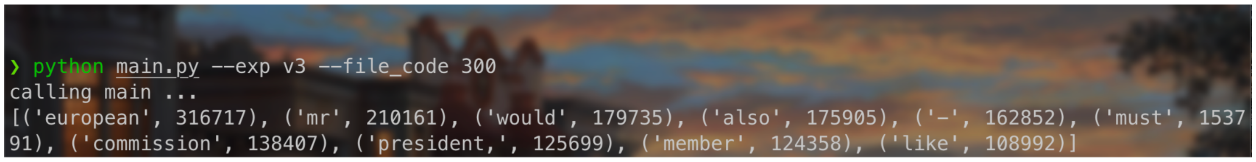
1. **Evaluation stability:**
   Since we ran all the experiments on a personal laptop instead of a dedicated server, there are many things that might affect the stability of the program, for example the current available ram, bg process running or even power levels.
2. **Resource utilization tracking:**
   We have not come up with the best practice of program performance tracking. In the single process version of code, the cProfile package works well, but when it comes to multiprocessing, the cross-processing performance tracking becomes really hard to analyze. So this might be one thing we need to further optimize.

Appendix - K Most Popular Words

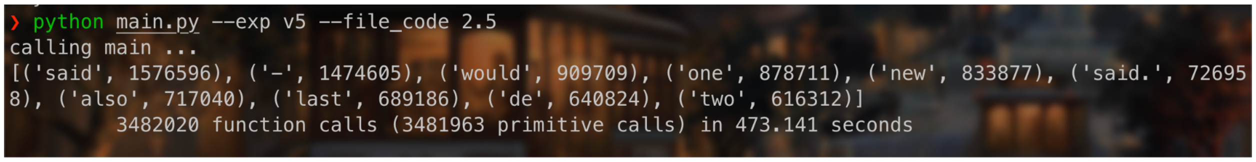
1. **50MB:**
   [('–', 45847), ('would', 21935), ('us', 18242), ('economic', 15389), ('new', 15365), ('one', 14078), ('countries', 13501), ('political', 12547), ('even', 12466), ('also', 12051)]
2. **300MB:**
   [('european', 316717), ('mr', 210161), ('would', 179735), ('also', 175905), ('-', 162852), ('must', 153791), ('commission', 138407), ('president,', 125699), ('member', 124358), ('like', 108992)]
3. **3.5GB**
   [('said', 1575355), ('-', 1473480), ('would', 908988), ('one', 878037), ('new', 833232), ('said.', 726393), ('also', 716495), ('last', 688663), ('de', 640313), ('two', 615817)]
4. **16GB**
   [('said', 10455872), ('would', 5794275), ('new', 5510611), ('one', 5390757), ('-', 5327377), ('said.', 4710820), ('also', 4552224), ('last', 4028819), ('two', 3842831), ('first', 3660769)]



```
❯ python main.py --exp v3 --file_code 50
calling main ...
[('–', 45847), ('would', 21935), ('us', 18242), ('economic', 15389), ('new', 15365), ('one', 14078), ('c
ountries', 13501), ('political', 12546), ('even', 12466), ('also', 12051)]
         389896 function calls (389880 primitive calls) in 3.113 seconds
```



```
❯ python main.py --exp v3 --file_code 300
calling main ...
[('european', 316717), ('mr', 210161), ('would', 179735), ('also', 175905), ('–', 162852), ('must', 1537
91), ('commission', 138407), ('president,', 125699), ('member', 124358), ('like', 108992)]
```



```
❯ python main.py --exp v5 --file_code 2.5
calling main ...
[('said', 1576596), ('–', 1474605), ('would', 909709), ('one', 878711), ('new', 833877), ('said.', 72695
8), ('also', 717040), ('last', 689186), ('de', 640824), ('two', 616312)]
         3482020 function calls (3481963 primitive calls) in 473.141 seconds
```