K Most Popular Word- RESULT ANALYSIS

- **Team members**
    1. Siva Sivasubramoniam Jayaram
    2. Reshma Subramaniam

- Code and Correct Output of code: https://github.com/SCU-Projects/KMostPopularWords

- Screenshots of results :
    - Output file attached for SINGLE threaded under /output folder
    - File names:
      output/[output_1gb.txt, output_8gb.txt, output_32gb.txt]
    - Note: Multi-threaded approach has exceptions while processing the data but the time to complete was validated correctly. Hence we did not include the output of it.

Algorithm & Data Structures & Design

- Algorithm**:**

```
func computeKMostPopularWords(){
   //disk read for buffer capacity
   //iterate across each line and update word-occurrence map
   //iterate across the keys in the map and update the PriorityQueue of size k
   //print output
}
```

- Data Structures:

1. Used Hashmap to store the word and its occurrence.
   Time: O(1) insert/update
   Space: O(n)

2. PriorityQueue of size k,
   Time: O(n*logk)
   Space: O(k)

- Design Pattern:

Producer/Consumer Multi-threaded.
Mode: Buffered-reader, NIO

Why this data-structure is better?

This algorithm uses two data structures namely a hashmap to store the word and its occurrence and a PriorityQueue to get the k frequent elements. It will perform better as it consumes less space and time compared to the other approaches.

For this problem space, with k = 100, this algorithm and data-structure suits best because of the following reasons:

1. Hashmap performs the getter and setter operations in O(1), so for a dataset of n words, the hashmap uses only O(n) time and space.
2. PriorityQueue consumes O(logk) for k insertion/update. For this problem space, since we are operating with n words, the total complexity is O(nlogk) which is good for large dataset.

Why this design is better?

We have used Multi-threaded Producer/Consumer approach for solving this problem space. Since the disk access is limited to only one I/O at any point of time, we cannot achieve speedup in the way disk access is performing. However, the computation of the word and its count map can be parallelized as it is performed in the regular computation. Hence a multi-threaded approach would give good performance speedup compared to a single-threaded approach.

In this problem space, we used a multi-threaded design to solve. It consists of 1 thread which will perform disk I/O similar to a Producer and the read disk data is processed by n consumers to populate the hashmap. We also observed a speedup of 26*60/170 => x 7.23(32 GB data).

Advantages of the data-structure and design w.r.t dataset size:

1. HashMap chosen has a linear time and space complexity for all data size.
2. PriorityQueue used uses constant space irrespective of the size of the data.
3. Producer/Consumer multi-threaded design has high performance over single-threaded application. Achieves x7 speedup compared to the single threaded application.

Presenting Performance Data:

S.No	Component	Single / Multi Theaded	Data (GB)	CPU utilization (%)	Memory utilization (GB)	Time to complete (seconds)	Speedup
1.	Buffered Reader	Single	1	185	2.00	26	1
			8	185	3.18	360	1
			32	185	3.66	1547	1
2.	NIO	Single	1	185	2.12	36	0.72
			8	185	3.18	450	0.80
			32	185	3.66	1900	0.81
3.	Buffered Reader	Multi	1	170	2.00	34	0.76
			8	290	2.12	202	1.78
			32	309	2.12	214	7.23

Explanation:

We processed all the 1, 8, 32 GBs of dataset with Single threaded design and using BufferedReaded and NIO. We observed that the program processed the dataset in a linear amount of time and NOT all the CPU cores were used efficiently. Hence, we switched to multi-threaded approach with a Producer Consumer design pattern.

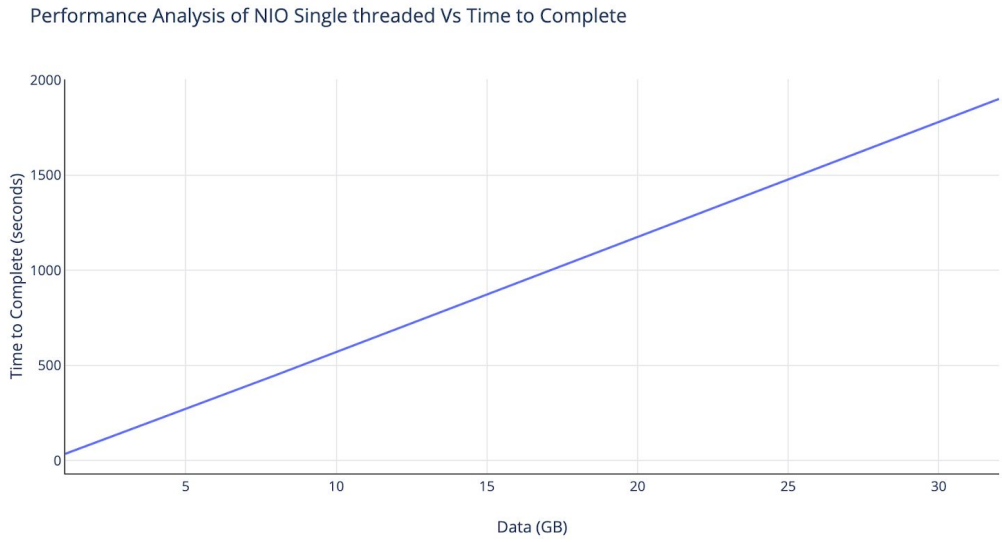
We were able to efficiently use all the CPU cores of the system. We approached the problem with allocating 1 thread for the Producer and 3 threads for the Consumer. We were able to achieve x7 speedup compared to the single-threaded application.

Below are the graphs explaining the performance of our application for different run configurations.

1. Performance Analysis of Buffered Reader Single threaded Vs Time to Complete

- Buffered Reader with single thread took 26 seconds for 1 GB data, 360 seconds for 8 GB data and 1547 seconds for 32 GB data.
  Verdict: No improvement on time to complete with single threaded buffered reader for all file sizes.

2. Performance Analysis of NIO Single threaded Vs Time to Complete

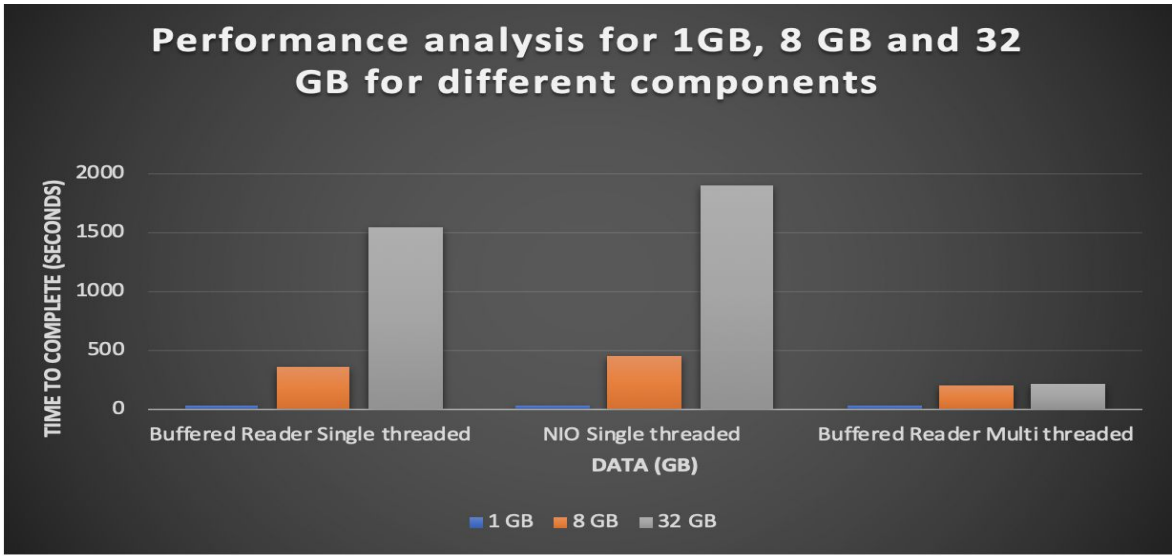


- NIO with single thread took 36 seconds for 1 GB data, 450 seconds for 8 GB data and 1900 seconds for 32 GB data.
  Verdict: No improvement on time to complete with single threaded nio buffered-reader for all file sizes.

3. Performance Analysis of Buffered Reader Multi threaded Vs Time to Complete

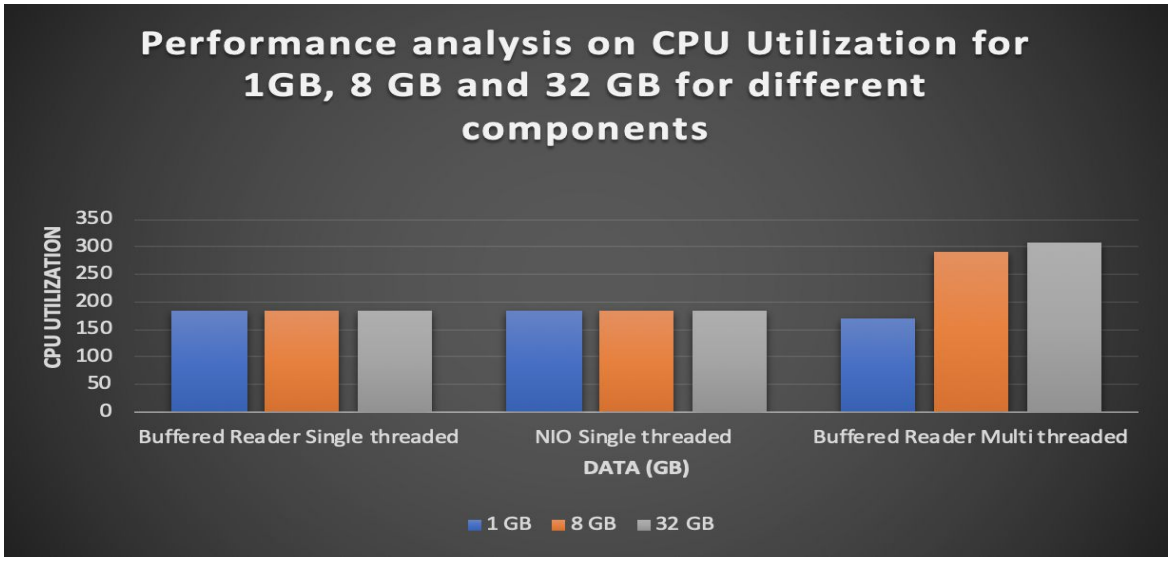
- Buffered Reader with multi threadr took 34 seconds for 1 GB data, 202 seconds for 8 GB data and **214 seconds for 32 GB data**. This is the most efficient algorithm we could come up for 32 GB data.
  Verdict: Good improvement on time to complete with multi threaded buffered reader for increasing file sizes.

4. Performance analysis for 1GB, 8 GB and 32 GB for different components



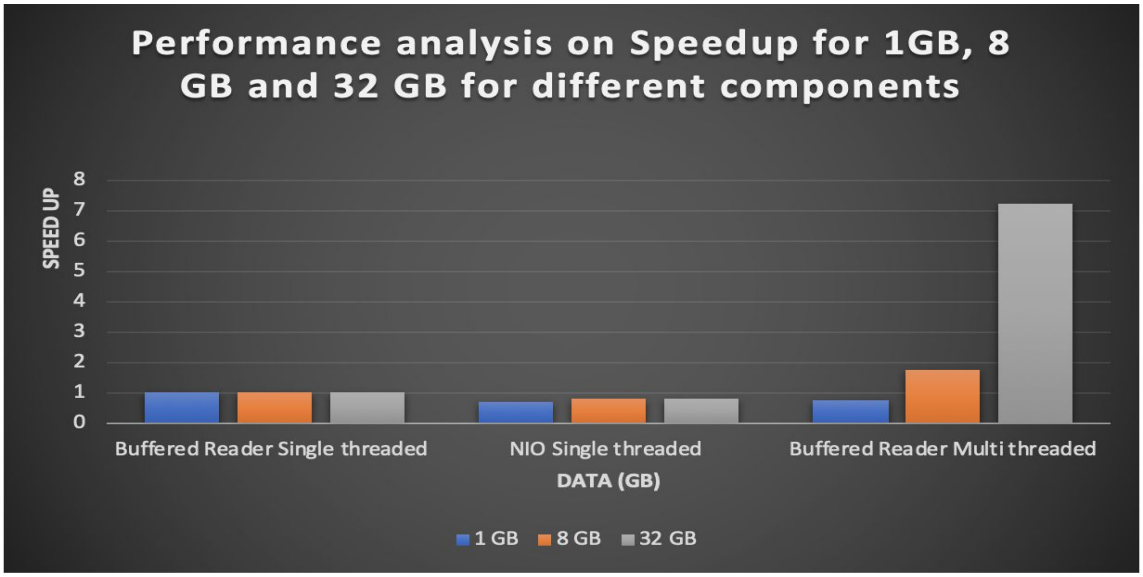
- Buffered reader with multi threading took 214 seconds to complete 32 GB data.
  Verdict: Multi threaded buffered reader performed the best compared to single threaded buffered reader and NIO based buffered reader for increasing file sizes.

5. Performance analysis for 1GB, 8 GB and 32 GB for CPU utilization



- CPU utilization is more as we change from single thread to multi thread for buffered reader.
  Verdict: Multi threaded buffered reader has good CPU utilization compared to single threaded buffered reader and NIO based buffered reader for increasing file sizes.

6. Performance analysis for 1GB, 8 GB and 32 GB for speedup



- We could achieve a speed up of factor 7.23 when using buffered reader multi threaded.
  Verdict: Multi threaded buffered reader has great speedup compared to single threaded buffered reader and NIO based buffered reader for increasing file sizes.

System specification:
RAM: 8GB, MacOS, MacBook Pro, 4 CPU cores

Techniques we should use to alleviate the problems introduced by increased input dataset size:

1. Improved algorithm
2. Increased heap size
3. Producer Consumer Multithreaded design pattern to improve throughput

Challenges Faced:
While running multi-threaded programs, the program is able to process 1GB data without any issues, however while running 8GB and 32GB dataset, we faced Garbage Collection exception. But we were able to observe that the program processed the entire dataset in 170 seconds. Because of the exception we are getting inconsistent output results which could be fixed by solving the Garbage collection exception.

Conclusion:

We analyzed three methods to populate most 100 frequently used words in a given data set. For 1 GB, single threaded with buffered-reader gave the best result. On increasing the size of data set, time taken to complete the task increased lineaerly for single threaded with buffered reader and NIO single threaded. Hence we came up with a multi threading approach with buffered reader to make the program more efficient. With this approach intially the program showed linear increase for 1 GB and 8GB data. After that for 32GB data we observed that there is a sharp decrease in the rate of growth due to the multiple threads into action and it stayed at an approximated constant time after that. We could achieve a speed up of factor 7.23 when using buffered reader multi threaded. With this we conclude that for computing k most frequent words for larger file sizes multi-threaded approach with producer consumer design pattern is a good fit.