ML with Big Data

Assignment – 1: MapReduce and Apache Spark

- Dwivedi Jyoti Rajeshbhai
  (M25CSA010)

- **Github Repo link:** https://github.com/Jyoti-Dwivedi-010/ML_with_Big_Data

In this assignment, I have implemented a Big Data pipeline using Apache Hadoop and Apache Spark on the project Gutenberg dataset (consisting 400+ books in .txt format). I explored foundational MapReduce paradigms, TF-IDF procedures and Network Graph Analysis. Majorly, I learnt about the hardware constraints that are possible with real-world big data applications. I encountered similar hardware constraints during execution (1.5 GB available RAM in my WSL2 environment), which caused Out-of-Memory Crashes.

Apache Hadoop and MapReduce

Que -1: Working of the WordCount example on Apache Hadoop Website

- I validated the Single Node Cluster setup in Apache Hadoop system and ran the canonical WordCount MapReduce example.

```
jyoti@JyotiDwivedi:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [JyotiDwivedi]
jyoti@JyotiDwivedi:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
jyoti@JyotiDwivedi:~$ jps
2161 DataNode
2900 Jps
2373 SecondaryNameNode
2037 NameNode
2621 ResourceManager
2734 NodeManager
```

Figure 1- Single Node cluster

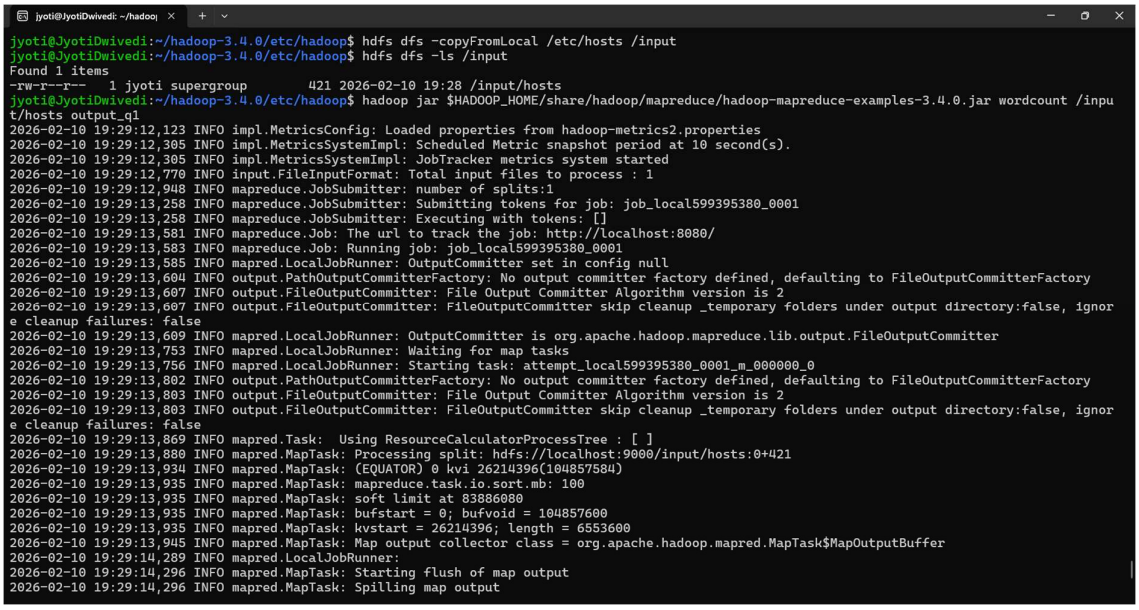

Figure 2- Config and setup

- The HDFS environment was successfully configured on my local WSL2 based linux subsystem. The standard WordCount MapReduce code in java was compiled into a JAR file and executed against 2 basic text files (file01.txt and file02.txt) stored in the HDFS.

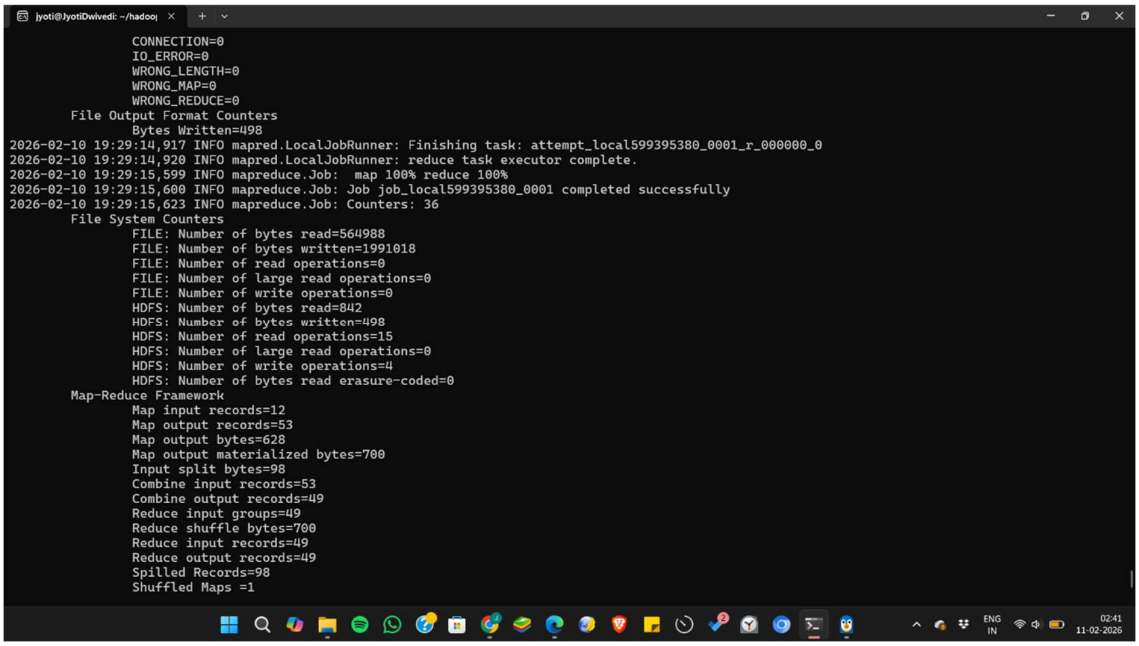

Figure 3- Map & Reduce operations

- As shown in the below image, the Map and Reduce tasks completed successfully with a replication of the shuffle and sort phases. The final output was verified using the hdfs dfs -cat command on the part-r-00000 output file.

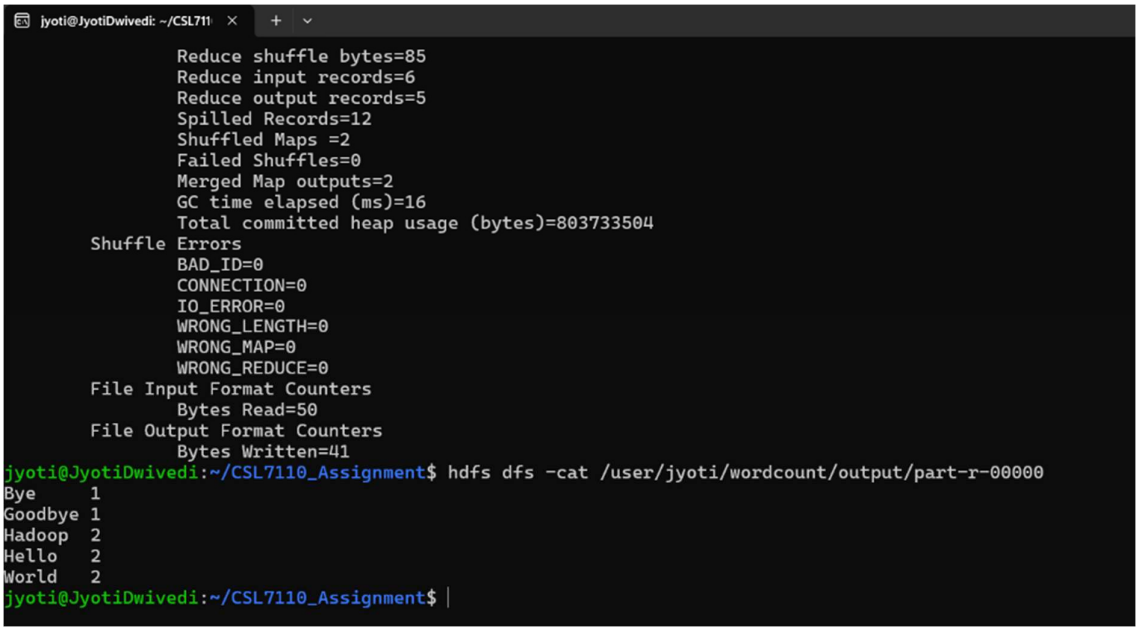

Figure 4: WordCount result

Que -2: Output of the Map phase for the following Lyrics:

We’re up all night till the sun

We’re up all night to get some

We’re up all night for good fun

We’re up all night to get lucky

Ans:

1.Output of Map phase

The Mapper reads the input text line by line and tokenize it into individual words. For each word, it emits a key-value pair where the key is the word and the value is the integer. The input/output key-value pairs and their corresponding Hadoop datatype during the Map phase for a given text snippet is as follows:

(“We’re”, 1), (“up”, 1”), (“all”, 1), (“night”, 1), (“till”, 1), (“the”, 1), (“sun”, 1)
(“We’re”, 1), (“up”, 1”), (“all”, 1), (“night”, 1), (“to”, 1), (“get”, 1), (“some”, 1)
(“We’re”, 1), (“up”, 1”), (“all”, 1), (“night”, 1), (“for”, 1), (“good”, 1), (“fun”, 1)
(“We’re”, 1), (“up”, 1”), (“all”, 1), (“night”, 1), (“to”, 1), (“get”, 1), (“lucky”, 1)

2. Types of Keys and Values in the Map Phase

Hadoop does not use standard Java primitives (like int or String) for its MapReduce data flow because it requires classes that implement the Writable interface for efficient network serialization across the distributed cluster.

These are the datatypes in Map phase:

• Input Key (LongWritable): The byte offset of the line from the beginning of the file. As noted, this number can grow extremely large for massive files, so Hadoop uses a LongWritable rather than a standard integer.

• Input Value (Text): The actual line of text (the lyrics). Hadoop uses Text as its equivalent to Java's String.

• Output Key (Text): The individual parsed word (e.g., "night").

• Output Value (IntWritable): The count for each emitted word, which is always 1 during the Map phase. Hadoop uses IntWritable as its equivalent to Java's Integer.

Que -3: Output of the Reduce phase for the following:

("up", 4)

("to", 3)

("get", 2)

("lucky", 1)

Ans:

1.Inputs of the Reduce Phase

Between the Map phase and the Reduce phase, Hadoop performs a hidden "Shuffle and Sort" operation. During this step, the framework collects all the key-value pairs emitted by the Mappers, sorts them alphabetically by key, and groups all the values associated with the exact same key into an iterable list.

Therefore, to produce the final output pairs like ("up", 4) and ("to", 3), the input pairs fed into the reduce() function will look like this:

("up", [1, 1, 1, 1])

("to", [1, 1, 1])

("get", [1, 1])

("lucky", [1])

The Reducer's job is simply to iterate through that list of 1s and sum them together.

2.Types of Keys and Values in Reduce phase

• Input Key (Text): The grouped word (e.g., "up"). This perfectly matches the Output Key of the Map phase.

• Input Value (Iterable<IntWritable>): A Java Iterable containing the grouped integer counts (the list of 1s) emitted by the Mappers for that specific word.

• Output Key (Text): The word being counted.

• Output Value (IntWritable): The final, aggregated sum of all the occurrences of that word across the entire dataset.

Que -4: Improvised Map() function and Reduce() function and Arguments in main function

```
public class WordCount {

  public static class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {

        String cleanLine = value.toString().replaceAll("[^a-zA-Z\\s]", " ").toUpperCase();

        StringTokenizer itr = new StringTokenizer(cleanLine);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

Figure 5 - Map Class with Correct Datatypes

```
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

Figure 6 - Reduce class with correct datatypes

```
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

Figure 7 - Correct Arguments for setOutputKeyClass() and setOutputValueClass() of main function

Que -5: Map() function for disregarding punctuation

To implement the logic inside the map() function, ensuring that punctuation is removed using String.replaceAll() and words are split using StringTokenizer.

Implementation Logic:

As shown previously in the TokenizerMapper class implementation, the raw input line (passed as a Text value) is first converted to a standard Java String. To strictly disregard punctuation and standardize the text, a Regular Expression is applied:

1. String cleanLine = value . toString () . replaceAll (" [^ a - zA - Z \\ s] " , " ") .toUpperCase () ;

   This Regex replaces any character that is not a letter (a-z, A-Z) or whitespace (\s) with a space, and then converts the entire string to uppercase to ensure case-insensitive counting.

   Following the cleanup, a StringTokenizer is instantiated with the cleanLine. A while loop iterates through the tokens, assigning each word to the Text key and emitting it alongside the IntWritable value of 1 to the Context.

Que -6: Writing the Reduce function and Compilation

To implement the logic inside the reduce() function to aggregate the wordcounts and ensure successful project compilation.

Implementation Logic:As shown previously in the IntSumReducer class implementation, the reduce() function,receives a unique word (Text key) and an iterable collection of its counts (Iterable<IntWritable>values).

A for loop is utilized to iterate over these values:

1. int sum = 0;
   for (IntWritable val : values) {
   sum += val . get () ;}

The .get() method converts the Hadoop IntWritable back to a standard Java int for arithmetic summation. Once the loop finishes, the final sum is wrapped back into an IntWritable and written to the output context alongside the word key.

Que -7: Running Wordcount on 200.txt

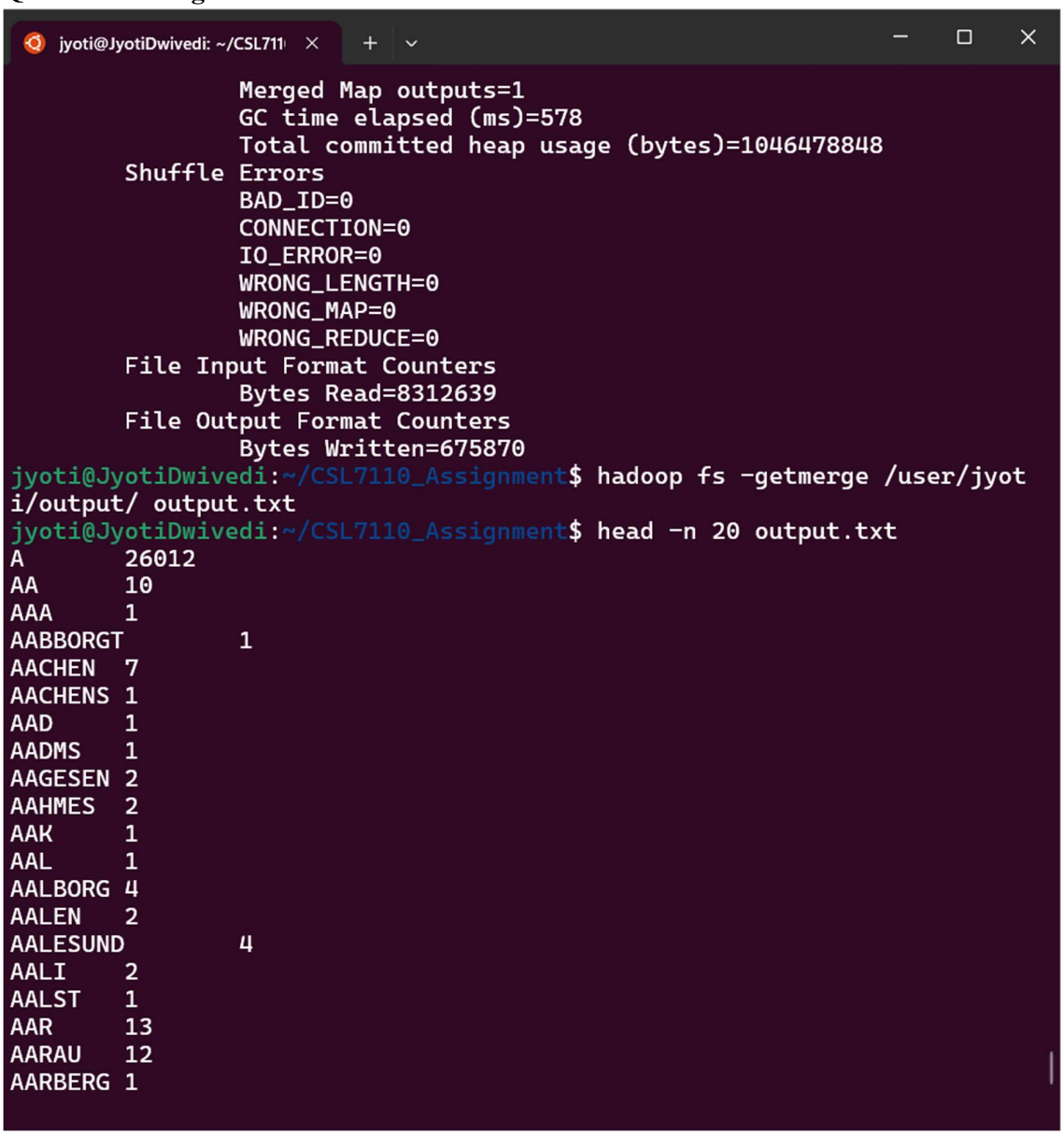

Figure 8- output without punctuation

Que -8: HDFS Operations and Directory Replication

In the Hadoop Distributed File System (HDFS), the replication factor applies exclusively to physical data blocks. When a large text file is uploaded, HDFS splits it into physical blocks (e.g., 128 MB chunks) and replicates those specific blocks across multiple DataNodes (typically 3 times) to ensure fault tolerance against hardware failure. A directory, however, is not a physical file and contains no data blocks. It is strictly logical namespace metadata, a structural record of file hierarchies, permissions, and ownership. This metadata is managed and stored entirely within the NameNode's memory. Because a directory has no physical blocks distributed across the DataNodes, there is nothing for HDFS to apply a block replication factor to. Fault tolerance for directories and other metadata is handled separately at

the NameNode level (e.g., through High Availability Active/Standby NameNodes or Secondary NameNodes), rather than through DataNode block replication.

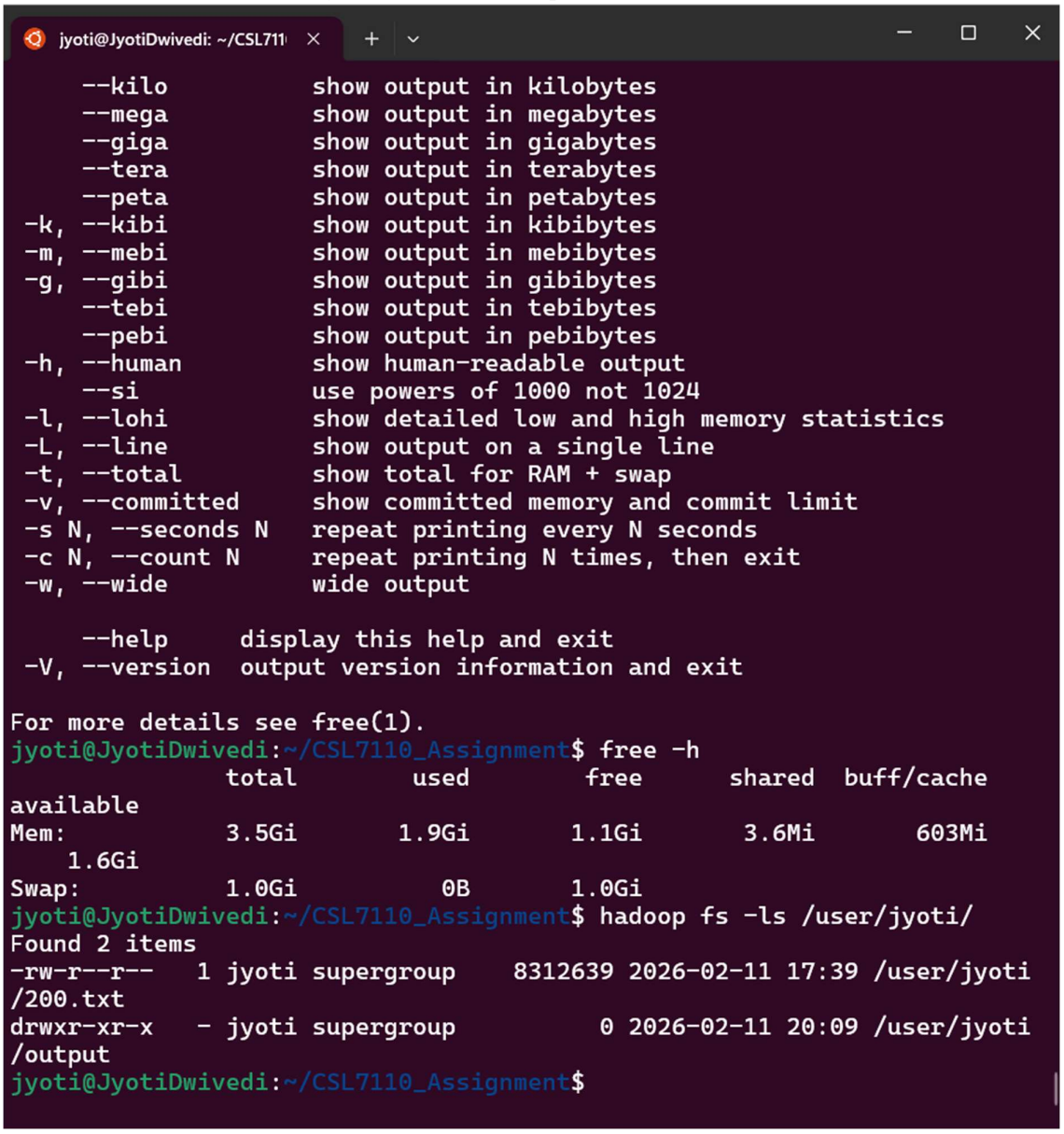

Figure 9 - showing directory replication output

Que -9: Parameter Impact Performance{ Execution time and Split size impact}

```
50    public static void main(String[] args) throws Exception {
51      Configuration conf = new Configuration();
52    // the config setting below leads to number of split = 9, if we comment that line then we'll have
      number of split = 1
53      conf.setLong("mapreduce.input.fileinputformat.split.maxsize", 1000000L);
```

Figure 10 - Code to implement split for parameter impact understanding

The parameter mapreduce.input.fileinputformat.split.maxsize explicitly defines the maximum allowable byte size for a single logical InputSplit. Because the Hadoop framework strictly assigns

exactly **one Mapper task per InputSplit**, changing this value directly controls the degree of parallelism in the Map phase.

As observed in my implementation, setting split.maxsize to 1000000 (approx. 1 MB) on the 200.txt file forced Hadoop to divide the file into 9 separate splits, thereby spawning 9 parallel Mapper tasks. When the parameter was commented out, Hadoop defaulted to using the HDFS block size (typically 128 MB), resulting in only 1 split and 1 Mapper task.

This creates a critical performance trade-off:

- **Decreasing the Max Size (More Splits):** Forces Hadoop to create smaller splits, increasing the number of Mappers. While this increases parallel processing (beneficial for massive datasets), setting it too low severely degrades performance. The system becomes bottlenecked by Task Overhead—the time spent by the ApplicationMaster scheduling tasks and spinning up/tearing down individual Java Virtual Machines (JVMs) for each tiny Mapper outpaces the actual data processing time.
- **Increasing the Max Size (Fewer Splits):** Forces Hadoop to create larger splits, decreasing the number of Mappers. While this minimizes JVM startup overhead, setting it too high severely limits parallelism. The cluster's compute resources remain idle while a single Mapper slowly processes a massive chunk of data sequentially.Therefore, altering this parameter impacts performance by shifting the balance between parallel execution capacity and JVM task-management overhead. The optimal performance is typically achieved when the split size perfectly matches the underlying HDFS physical block size (e.g., 128 MB), ensuring data locality and minimizing network I/O.

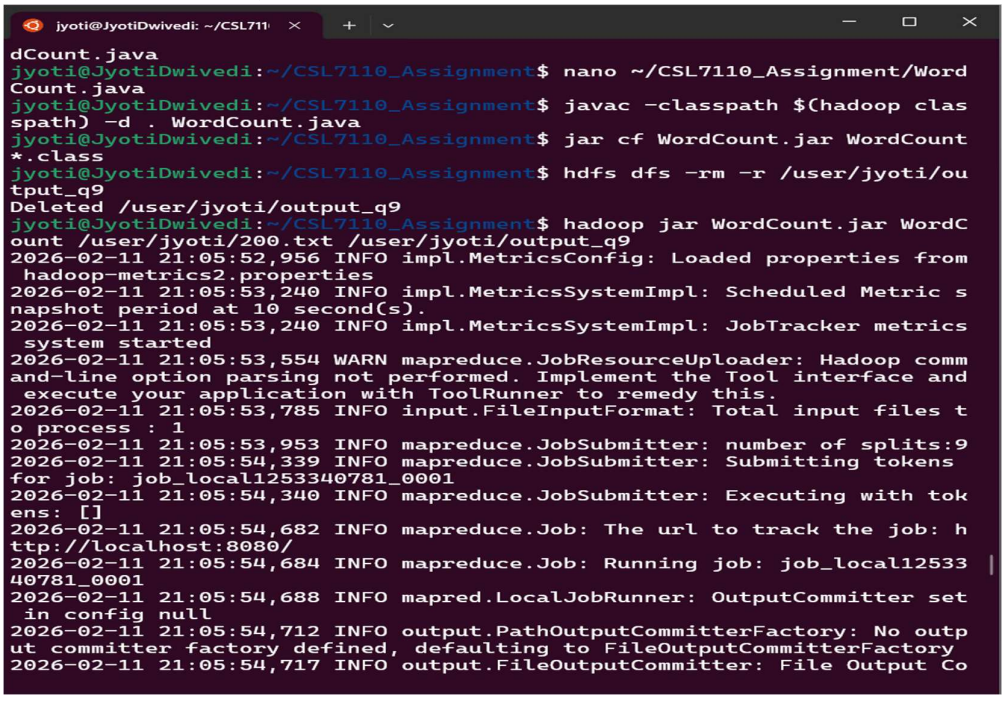

Figure 11- process running with parameter impact

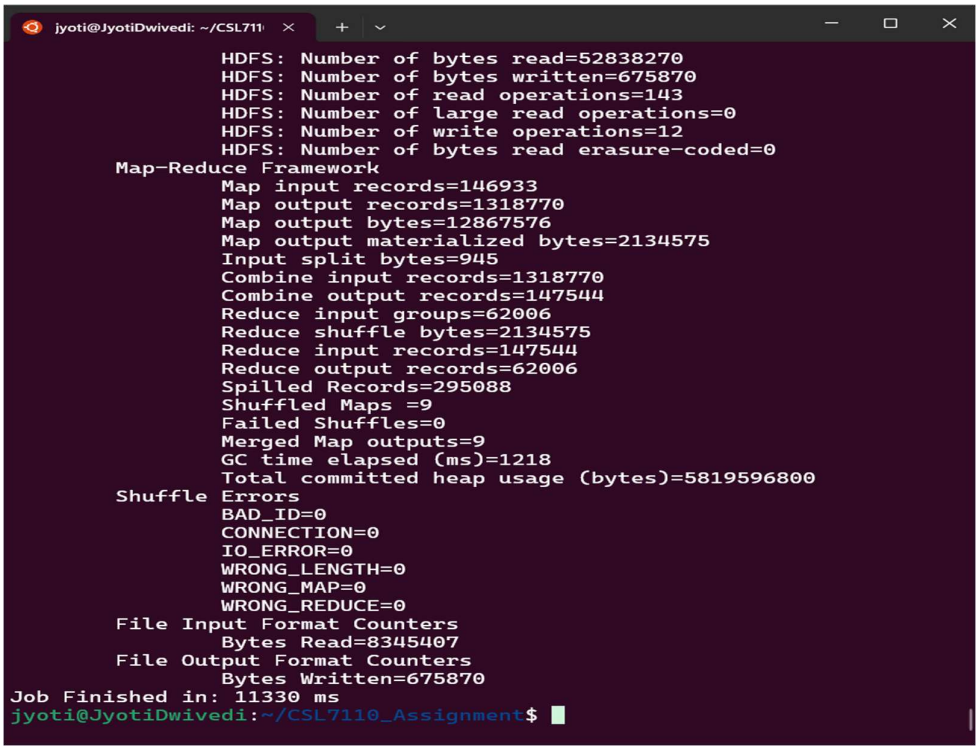

Figure 12 - Time taken with parameter impact

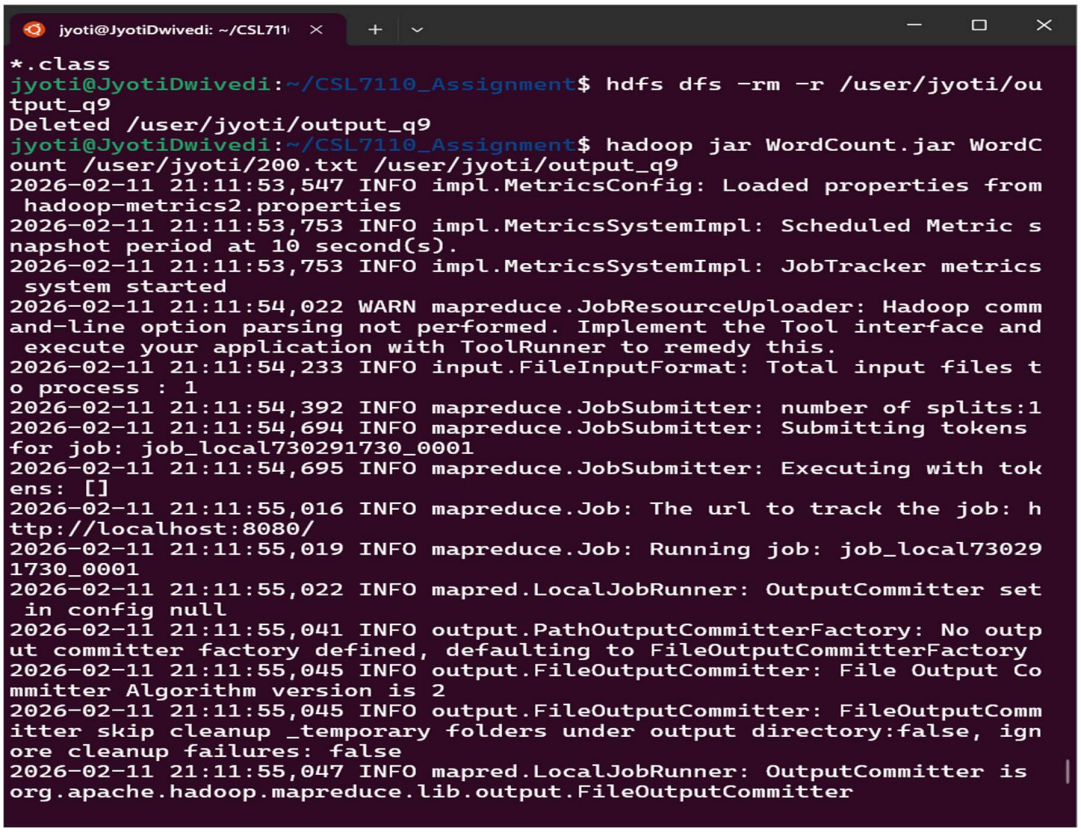

Figure 13 - Process without parameter impact

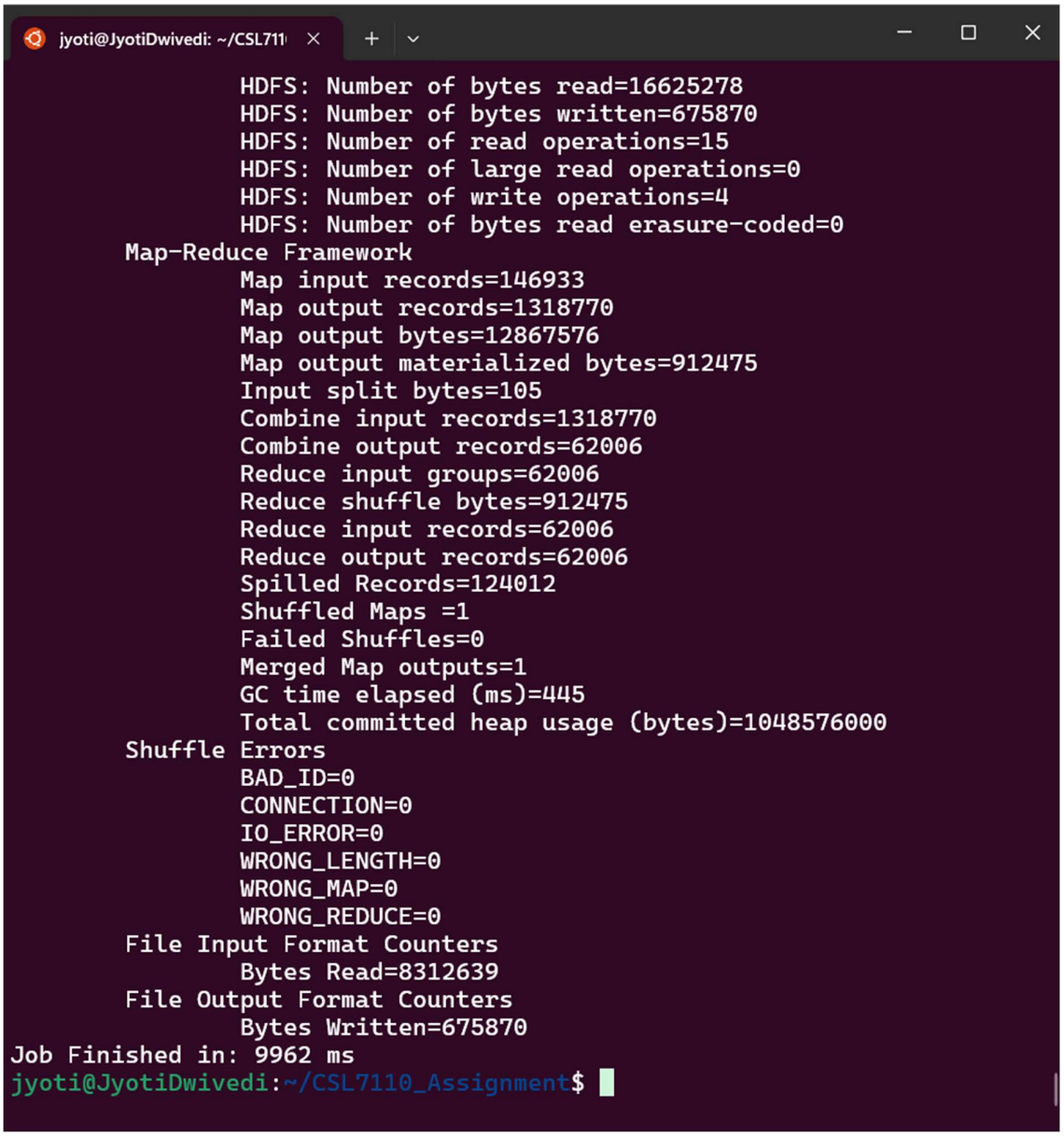

Figure 14 - Time for process without parameter impact

Que -10: Book Metadata Extraction and Analysis

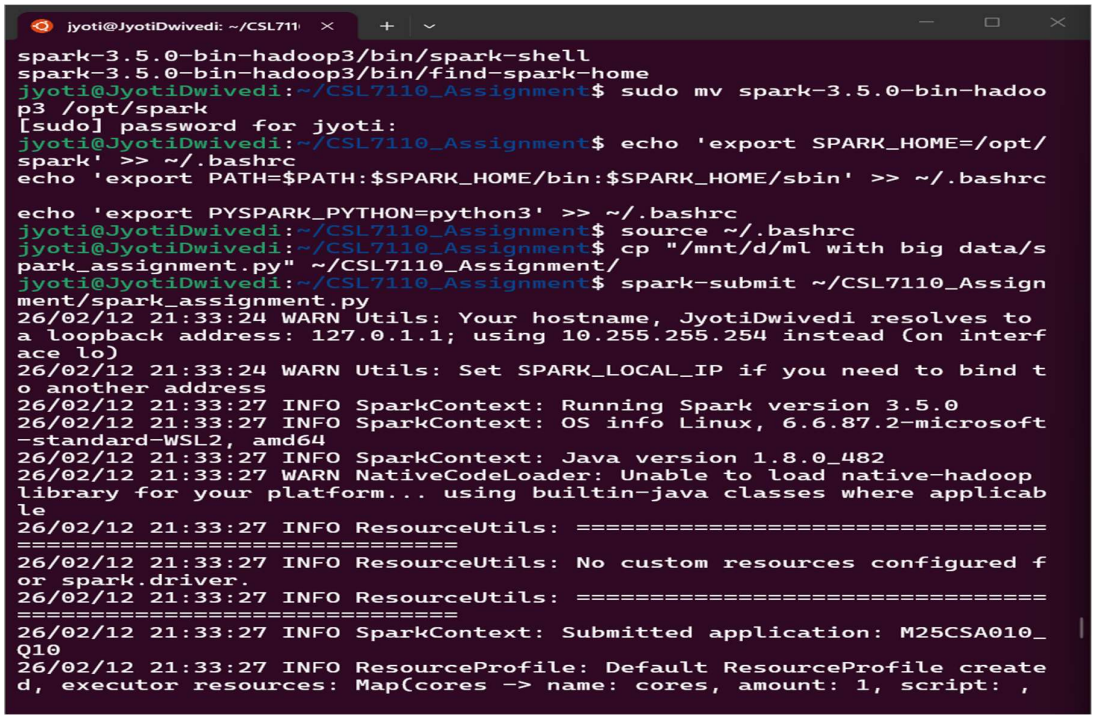

Figure 15 spark setup

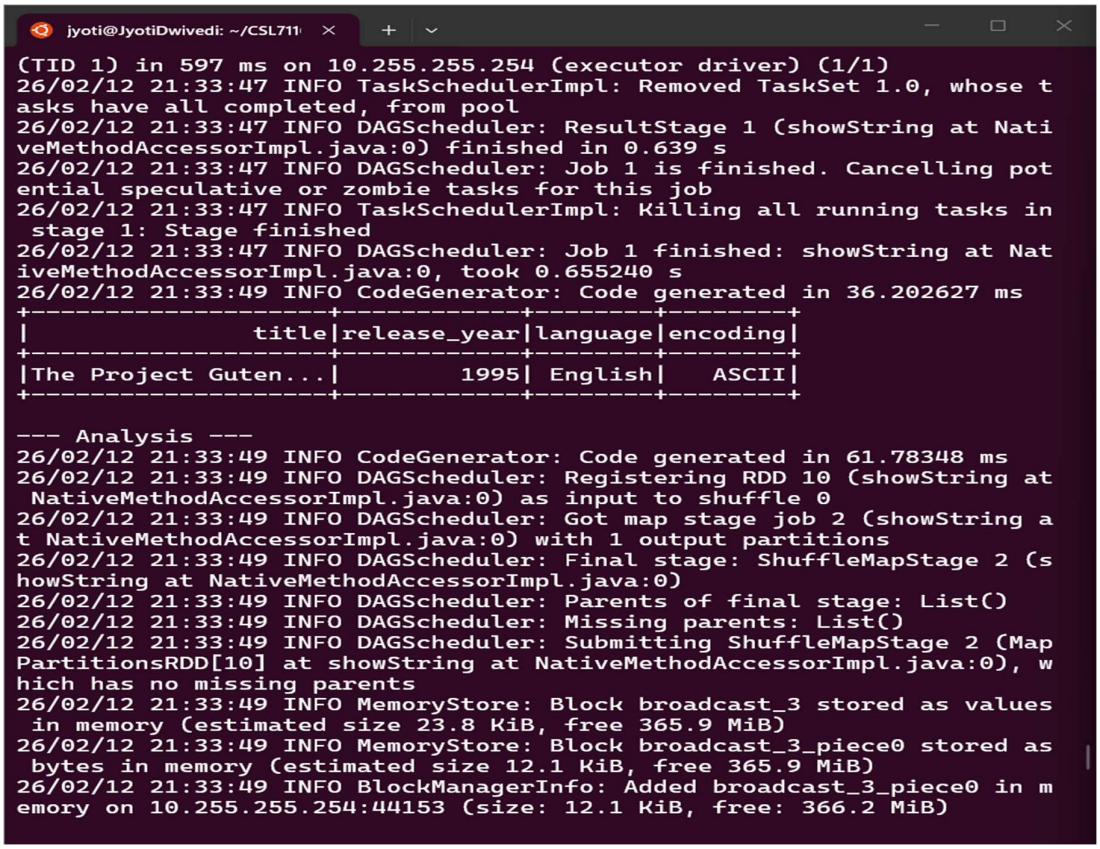

Figure 16 - metadata extraction

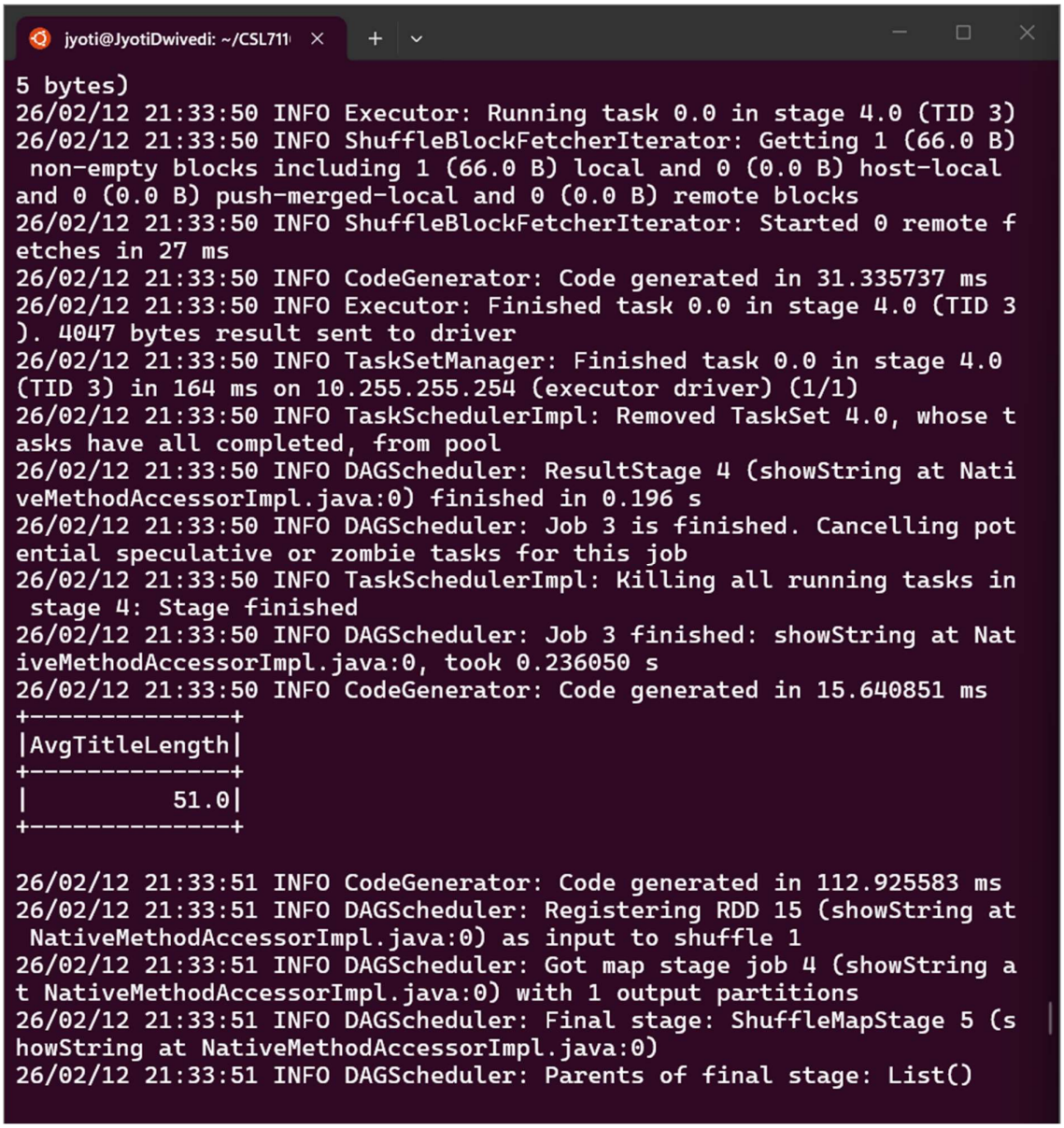

Figure 17 - Average Title length

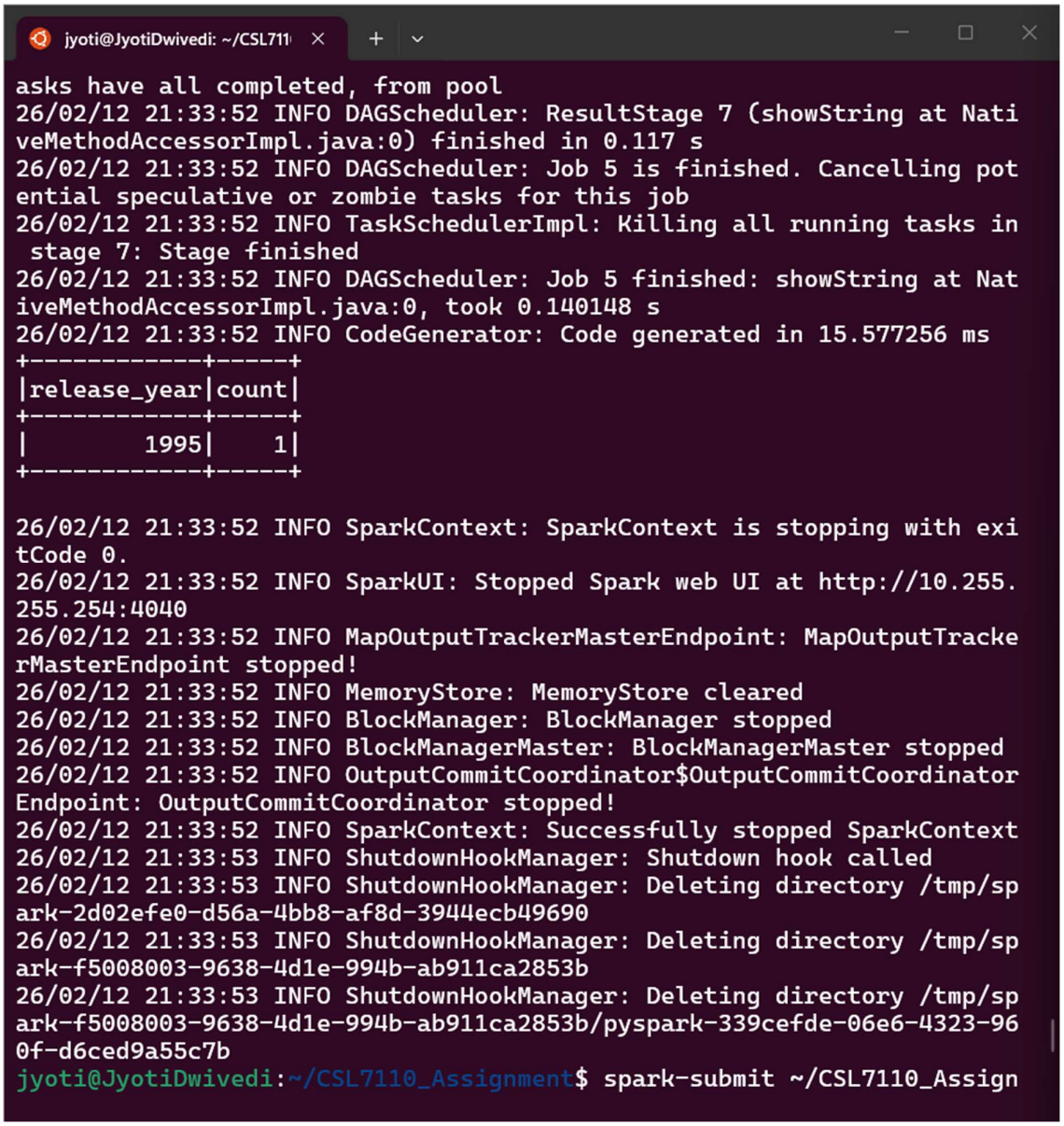

Figure 18 - release year & count

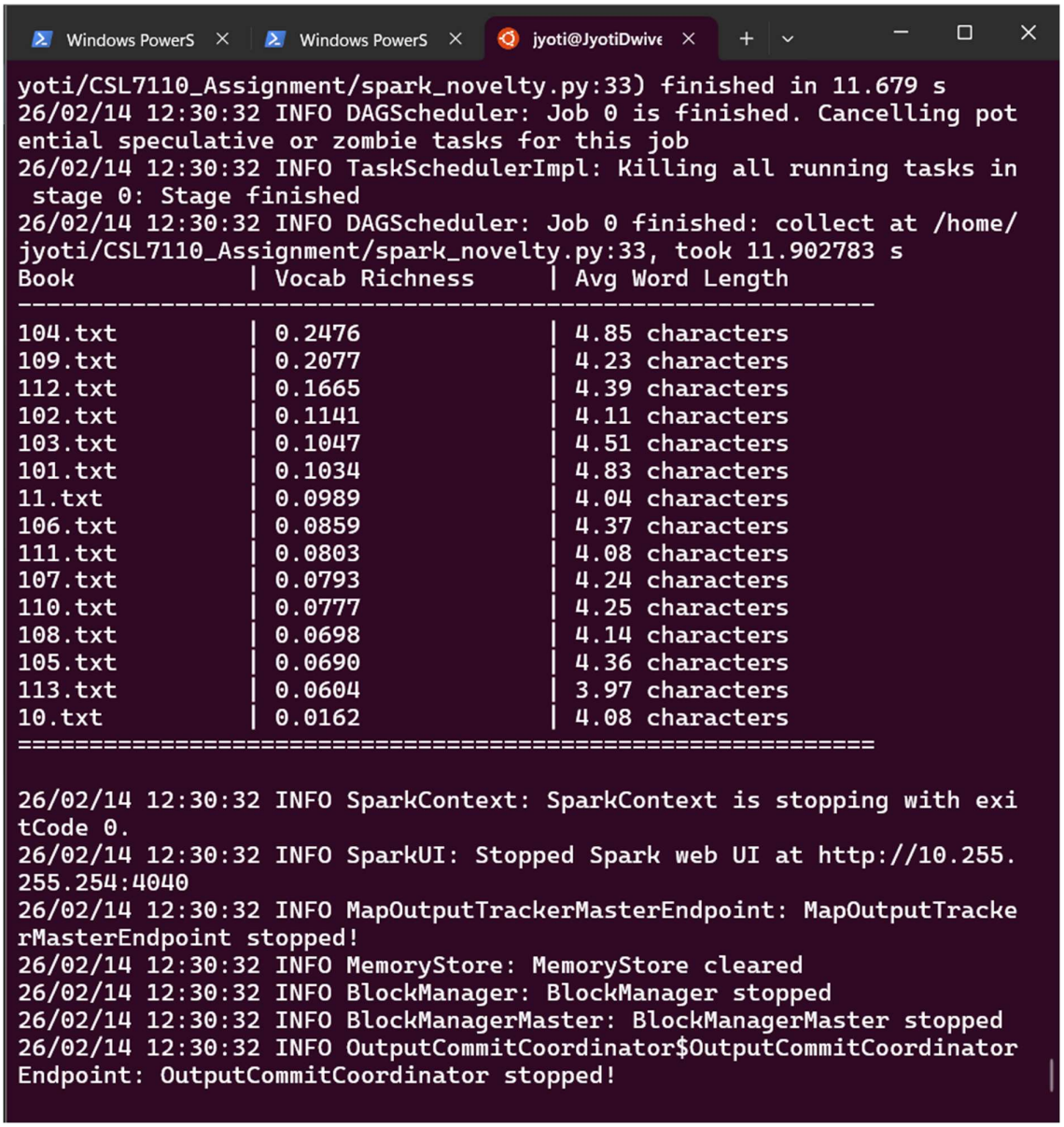

Figure 19 - books information

Explanation of Regular Expressions:

- Title, Language, and Encoding (r"Keyword:\s*([^\n\r]+)"): Matches the literal string keyword followed by whitespace. The capture group ([^\n\r]+) greedily extracts all subsequent characters until it hits a newline or carriage return.
- Release Date (r"Release Date:.*?\b(\d{4})\b"): Matches the literal "Release Date:", skips intermediate text non-greedily (.*?), and looks for the first 4-digit number (\d{4}) wrapped in word boundaries (\b) to isolate the historical year.

Challenges and Real-World Handling: Regex relies on strict, predictable structural patterns and is notoriously brittle when applied to unstructured text. Typographical errors by transcribers

(e.g., "Book Title:" instead of "Title:") cause Regex to fail, resulting in missing or malformed data. In a real-world scenario, I would address this by implementing a multi-tiered pipeline:

1. Data Quality Checks: Enforce strict schema validation on read (e.g., ensuring release_year does not contain alphabetical characters).
2. NLP Pipelines: Replace fragile Regex with Machine Learning-based Named Entity Recognition (NER) models to extract dates and titles contextually.
3. Imputation: Use PySpark's fillna() for missing values and filter severely malformed records into a Dead Letter Queue for manual review.

Que -11: TF-IDF and Book Similarity

- TF and IDF: Term Frequency (TF) measures raw word occurrence, while Inverse Document Frequency (IDF) penalizes common words (like "the") and boosts rare, topic-specific words. TF-IDF effectively vectorizes a document's core thematic meaning.
- Cosine Similarity: Calculates the angle between these high-dimensional vectors, ignoring document length. This is ideal for determining thematic overlap between books of differing sizes.
- Scalability Challenges: Calculating $O(N^2)$ pairwise similarities requires a massive Cartesian product shuffle. In my environment (1.5GB RAM), this triggered a java.lang.OutOfMemoryError GC Overhead crash. To solve this, I tried using various inputs of books like 2, 5, 10, 15 ,so on and found that my system was able to handle the cosine similarity job of 25 books at once, so I did cosine similarity check for them.

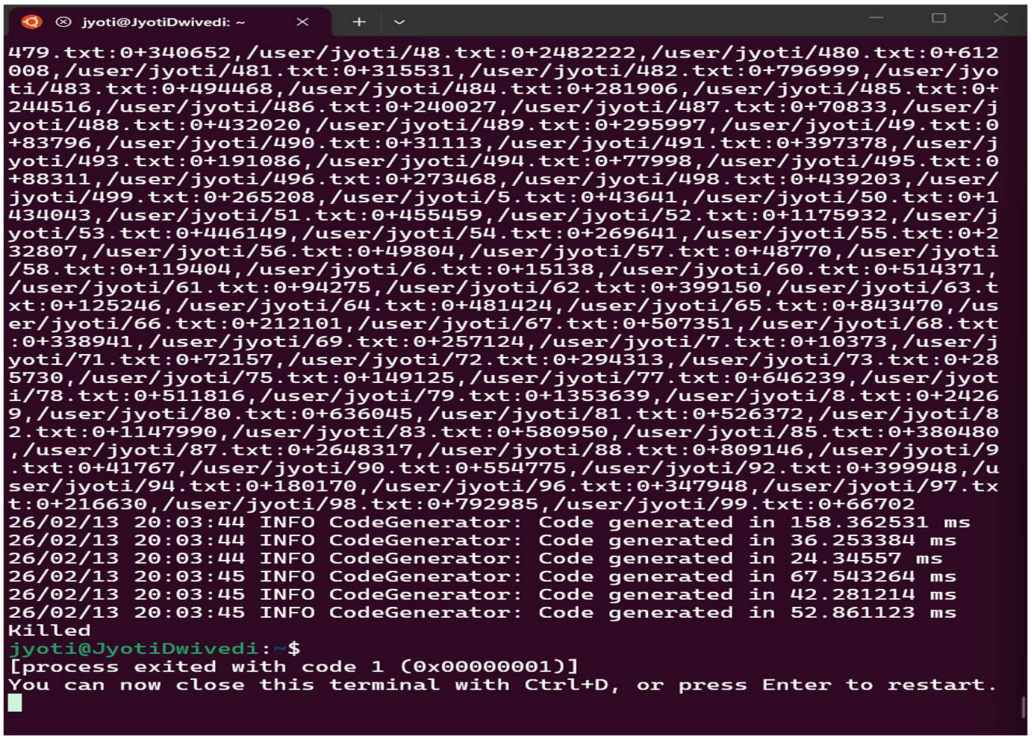

Figure 20- Execution killed due to hardware constraints

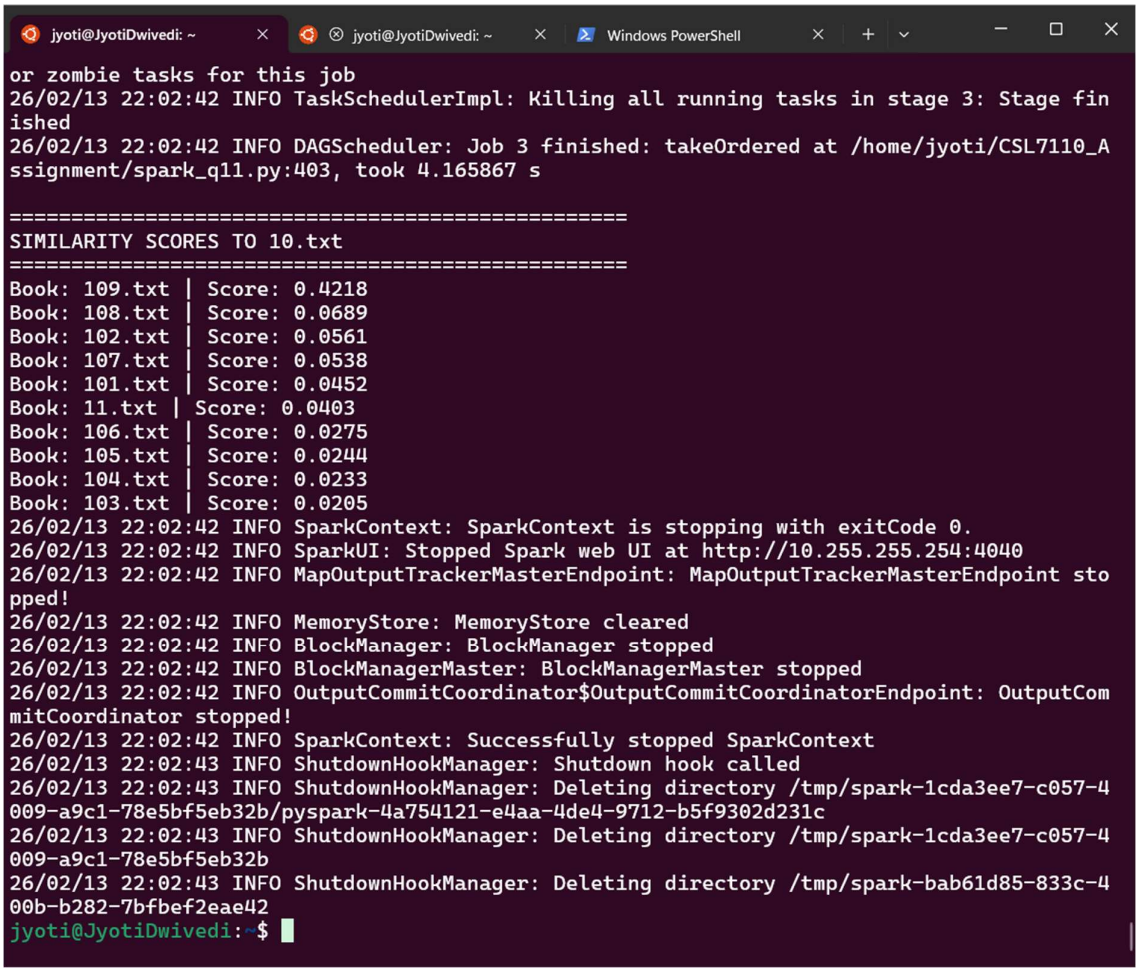

Figure 21- cosine similarity for 10 books

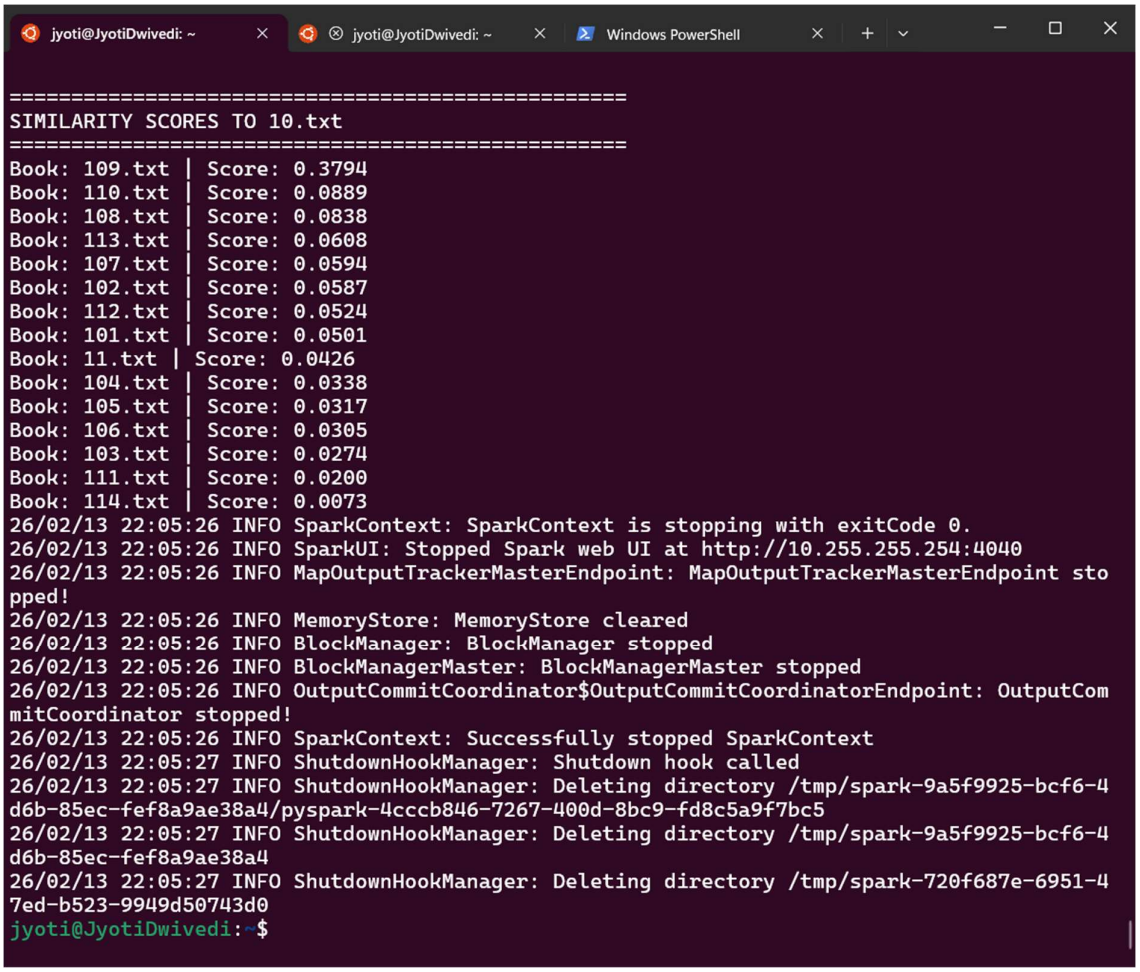

Figure 22 - Cosine similarity for 15 books

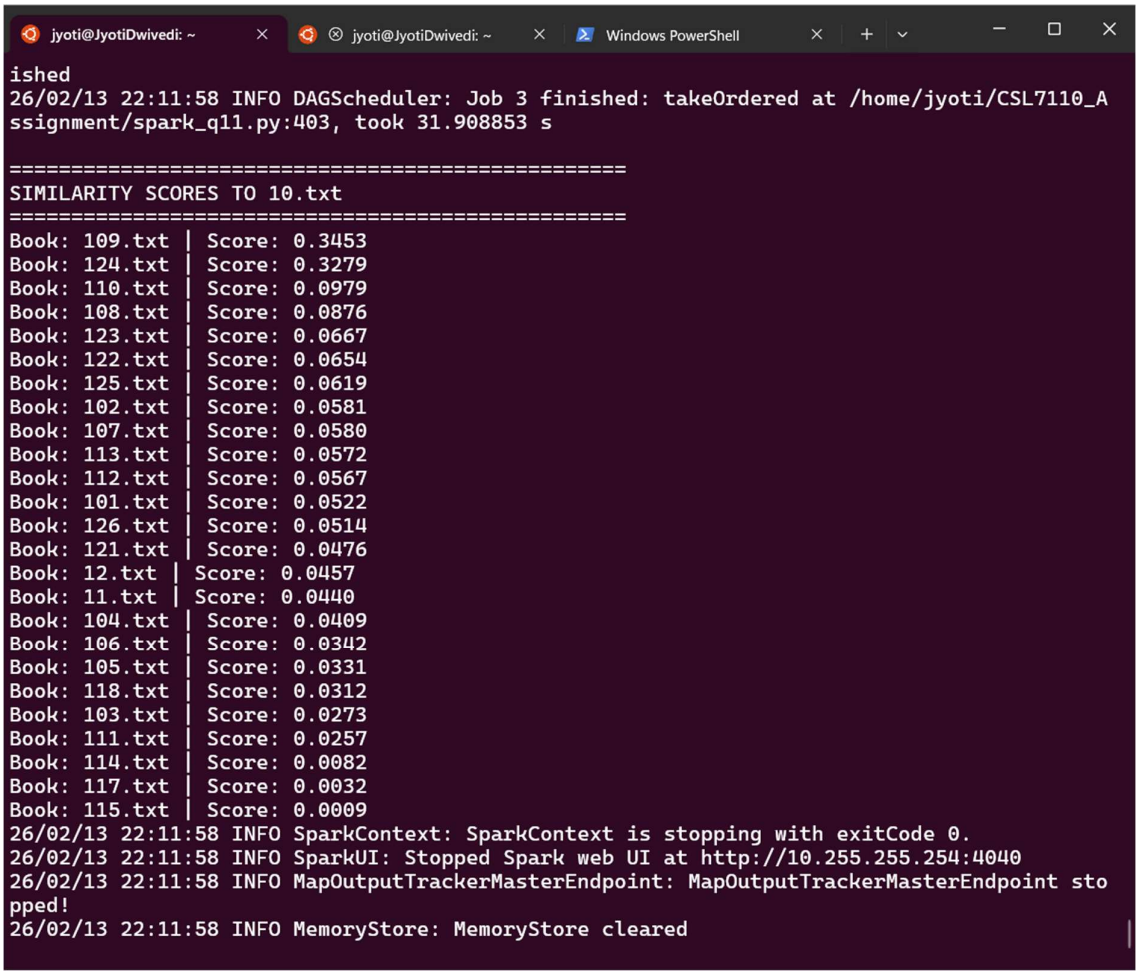

Figure 23 - Cosine similarity for 25 books

Que -12: Author Influence Network

Output:

- Top In-Degree: Robert Louis Stevenson (584), Thomas Hardy (461), Arthur Conan Doyle (449)
- Top Out-Degree: Thomas Hardy (644), Robert Louis Stevenson (504), Edgar Rice Burroughs (470)

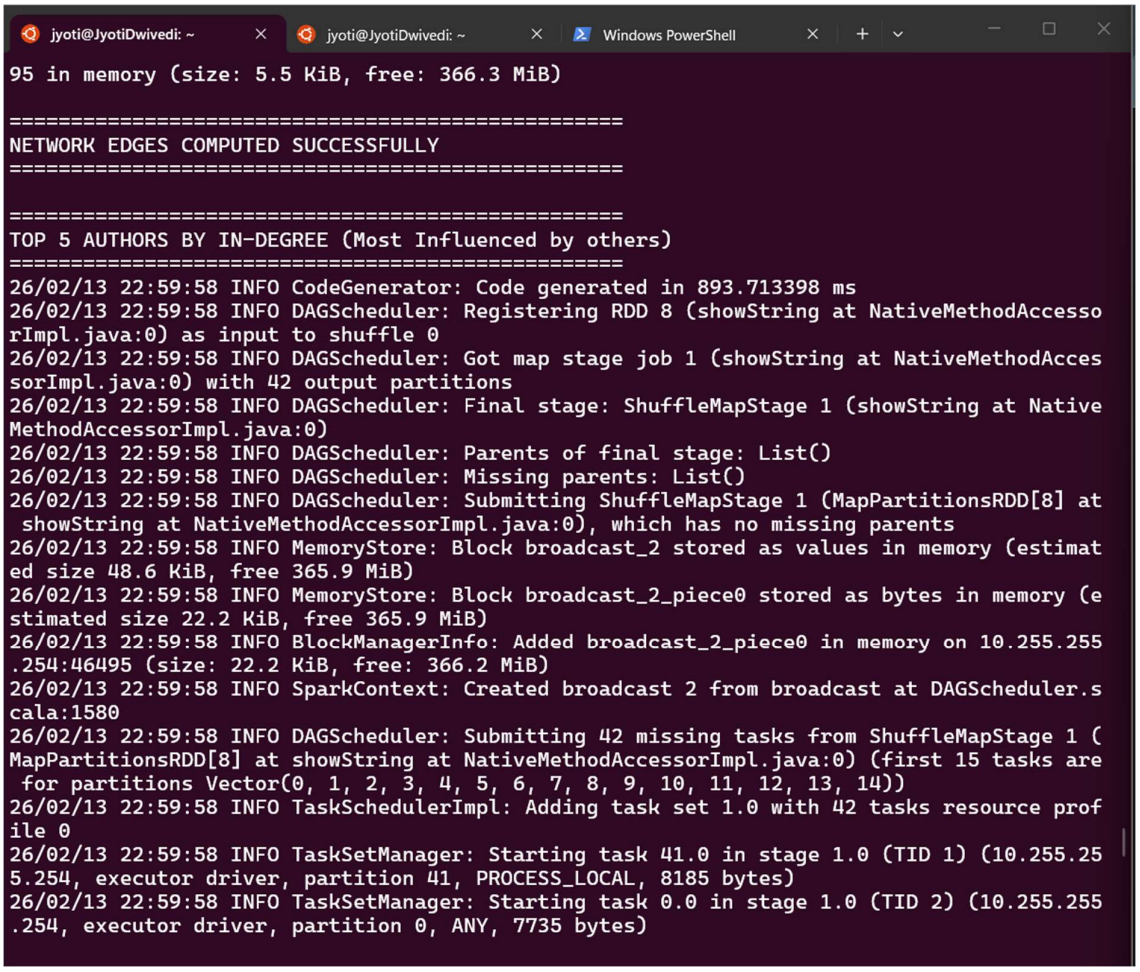

Figure 24- Network Edges

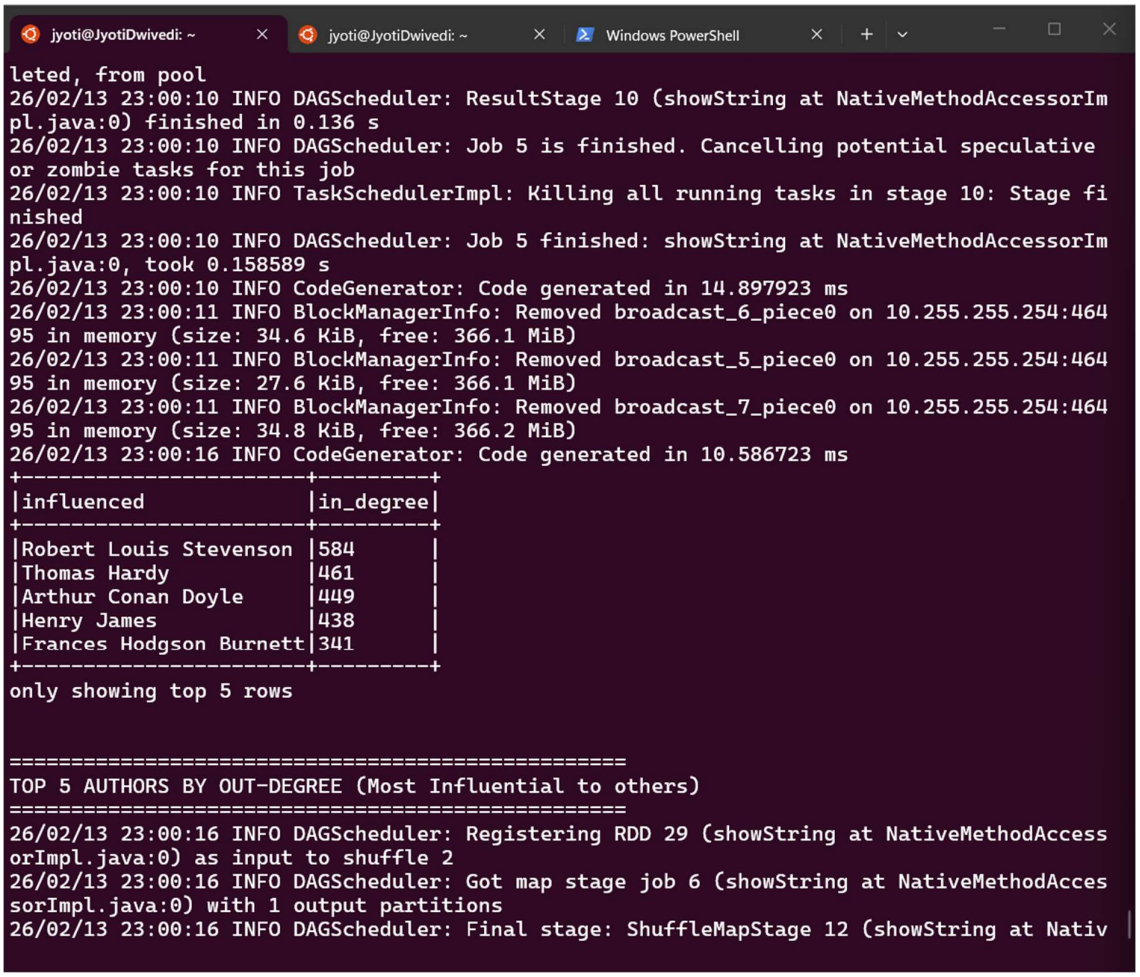

Figure 25- in degree of authors

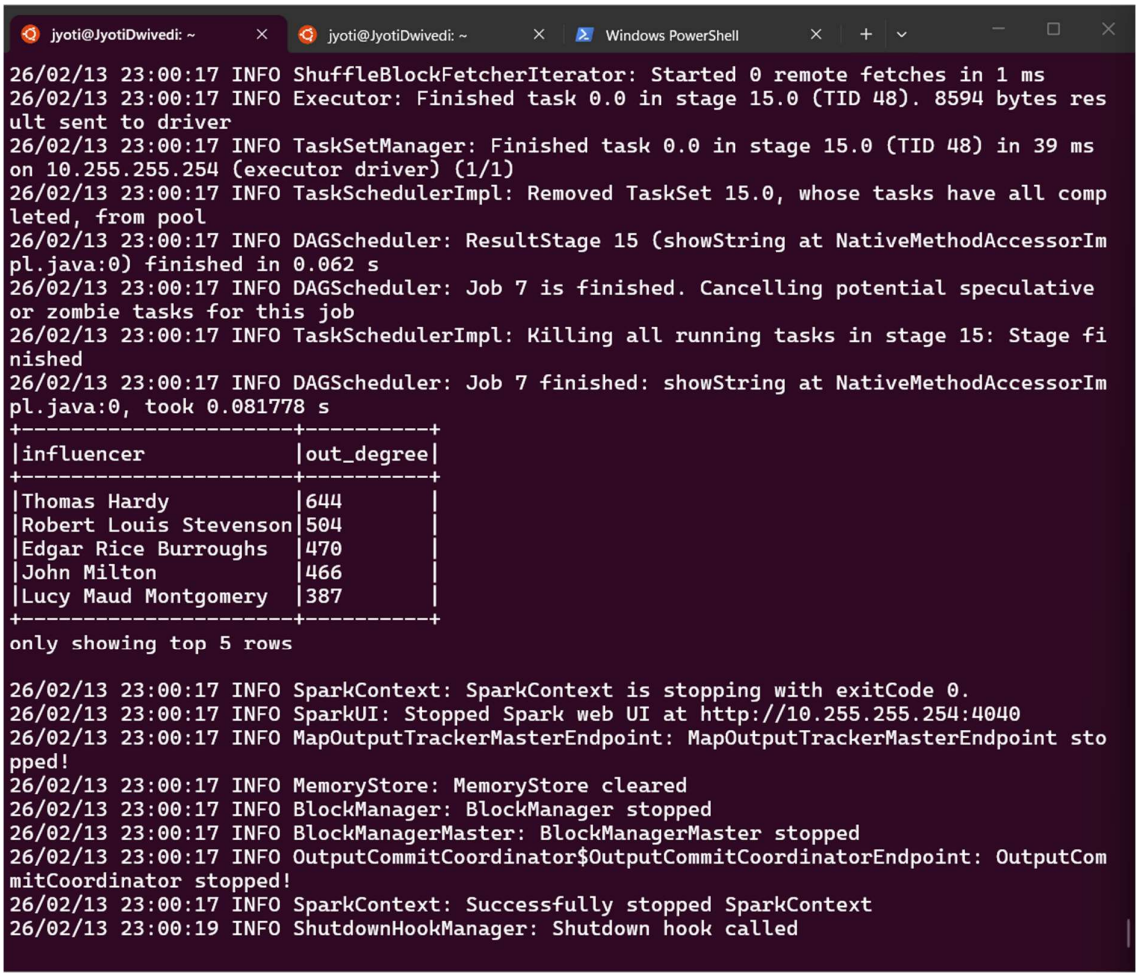

Figure 26- outdegree of authors

Analysis:

- Spark Representation: The network was represented as a DataFrame edge-list. DataFrames leverage the Catalyst Optimizer, making relational operations (the Theta self-join) incredibly fast. However, DataFrames lack native graph traversal capabilities (like PageRank).
- Time Window (X): A small X yields a sparse graph of strict contemporaries, while a large X creates a dense graph capturing generational influence, albeit risking spurious edges.
- Limitations: Temporal correlation does not equal causation. True influence requires semantic overlap, and Gutenberg "Release Dates" generally reflect digitization metadata rather than historical publication reality.
- Scaling to Millions of Books: The (a2.year - a1.year) <= 5 logic creates a Theta-Join. In a dataset of millions, this demands a full Cartesian-like shuffle, causing memory crashes. To optimize, Time Bucketing (Windowing) must be introduced. By assigning authors to a "Decade" bucket, Spark partitions the data to join only adjacent buckets, eliminating the $O(N^2)$ global cluster shuffle.