CSL7110 Assignment 1
MapReduce and Apache Spark

S Kartik Iyer
M25CSA025

Contents

1 Hadoop and MapReduce

1.1 WordCount Execution

The WordCount example was successfully executed on a single-node Hadoop setup. The output showed correct word frequency counts for the input file.

The Map phase emitted pairs of the form:

$$(word, 1)$$

The Reduce phase grouped values and produced final counts:

$$(word, total\_count)$$

This verified that the Hadoop setup was functioning correctly.

1.1.1 Explanation of Map Phase Output (Question 2)

In my implementation, I used StringTokenizer to perform the tokenization of each input line.During the Map phase, each input line is tokenized into individual words. For every word encountered, the mapper emits an intermediate key-value pair of the form:

$$(word, 1)$$

Example mapper outputs include:

```
("we’re", 1)
("up", 1)
("all", 1)
("night", 1)
("to", 1)
("get", 1)
("some", 1)
("for", 1)
("good", 1)
("fun", 1)
("lucky", 1)
```

Repeated occurrences of a word generate multiple intermediate pairs. The mapper receives input of type (LongWritable, Text), where the key represents the byte offset of a line and the value contains the full text line. The mapper outputs pairs of type (Text, IntWritable).

After mapping, Hadoop performs shuffle and sort operations which group all values corresponding to identical keys before sending them to the reducer.

1.1.2 Explanation of Reduce Phase Input and Output (Question 3)

After the shuffle and sort phase, the reducer receives grouped key-value pairs where each key corresponds to a unique word and the associated iterable contains all occurrences of that word generated during mapping.

Example reducer input:

```
("up", [1, 1, 1, 1])
("to", [1, 1, 1])
("get", [1, 1])
("lucky", [1])
("night", [1, 1, 1, 1])
("we’re", [1, 1, 1, 1])
```

The reducer input has type:

(Text, Iterable¡IntWritable¿)

The reducer iterates through the iterable, sums the values, and emits the final word count. The reducer output type is:

(Text, IntWritable)

Example reducer output:

```
("up", 4)
("to", 3)
("get", 2)
("lucky", 1)
("night", 4)
("we’re", 4)
```

1.2 Map and Reduce Data Types

Map Input: (LongWritable, Text)

Map Output: (Text, IntWritable)

Reduce Input: (Text, `Iterable<IntWritable>`)

Reduce Output: (Text, IntWritable)

1.3 Effect of Split Size Parameter

Changing `mapreduce.input.fileinputformat.split.maxsize` directly affected the number of map tasks.

Smaller split size:

- More map tasks
- Higher parallelism
- Increased overhead

Larger split size:

- Fewer map tasks
- Reduced overhead
- Limited parallelism

Optimal performance required balancing these two factors.

2 Apache Spark Setup

Apache Spark was installed in local mode.

Initially, Spark attempted to connect to HDFS, resulting in repeated errors:
`java.net.ConnectException: Connection refused`
The permanent solution was:

```
pyspark --master local[2] \
--conf spark.hadoop.fs.defaultFS=file:///
```

This forced Spark to use the local filesystem instead of HDFS. After this change, all file-loading errors were resolved.

3 Question 7

The file `200.txt` was copied from the local filesystem to HDFS using the `hdfs dfs -put` command. The WordCount MapReduce job was executed successfully on this dataset. The output results were stored in HDFS and verified using commands such as `hdfs dfs -cat` and `hdfs dfs -ls`.

4 Question 8

4.1 Replication Factor in HDFS

In HDFS, replication is applied only to files and not to directories. Directories do not store actual data blocks; instead, they maintain metadata such as file names, ownership, permissions, and directory hierarchy. This metadata is managed by the NameNode.

Files are divided into blocks and stored across multiple DataNodes. These blocks are replicated to provide fault tolerance, reliability, and high availability of data. If one DataNode fails, data can still be retrieved from other replicas.

Since directories do not contain data blocks, there is no physical data to replicate. Directory safety is ensured through NameNode metadata mechanisms such as FsImage snapshots and edit logs. In distributed deployments, High Availability configurations with standby NameNodes further improve reliability.

5 Question 10: Metadata Extraction

5.1 Approach

The dataset contained raw Gutenberg books. Metadata fields were extracted using regular expressions:

- Title: `Title:\s*(.*)`
- Release Date: `Release Date:\s*(.*)`
- Language: `Language:\s*(.*)`
- Encoding: `Character set encoding:\s*(.*)`

5.2 Challenges Faced

- Inconsistent metadata formatting
- Some books missing language field
- Release dates written in different formats
- Duplicate header lines

Initially, the most common language was detected as empty string due to missing values. After filtering out empty entries, English appeared as the most common language.

5.3 Insights

Regular expressions work only when formatting is consistent. In real-world systems, structured metadata storage is preferable.

6 Question 11: TF-IDF and Similarity

6.1 Preprocessing

The following steps were performed:

- Converted text to lowercase
- Removed punctuation
- Tokenized into words
- Removed stopwords

6.2 TF-IDF

Term Frequency:

$$TF(w, d)$$

Inverse Document Frequency:

$$IDF(w) = \log \frac{N}{DF(w)}$$

TF-IDF:

$$TF\text{-}IDF(w, d) = TF(w, d) \times IDF(w)$$

6.3 Cosine Similarity

$$Similarity(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

The top 5 books most similar to `10.txt` were identified. The highest similarity score was approximately 0.86.

6.4 Major Debugging Issues

During similarity computation, I encountered:

- Py4JJavaError
- Java gateway empty response
- Memory crashes

Cause: Large joins and Python UDF overhead.
***Fix* :**

- Reduced dataset size before similarity computation
- Avoided unnecessary collect() operations
- Ensured Spark ran in local mode

This significantly stabilized execution.

7 Question 12: Author Influence Network

7.1 Preprocessing Improvements

Initially, I attempted to use:
`wholeTextFiles()`
This caused deadlock errors:
`Detected deadlock while completing task`
Reason: Entire 184MB dataset was loaded into memory.
Fix:

- Switched to `spark.read.text()`
- Limited to header lines only
- Extracted only required metadata fields

This reduced memory usage drastically.

7.2 Network Representation

I represented the network as a Spark DataFrame:

$$(author1, author2)$$

where author1 potentially influenced author2.
I chose DataFrame instead of RDD because:

- Easier joins
- Optimized by Catalyst engine
- Cleaner aggregation

7.3 Errors Faced and Fixes

1. HDFS Connection Error
Fixed by forcing local filesystem configuration.
2. Deadlock using wholeTextFiles
Fixed by switching to line-by-line text loading.
3. Java Gateway Empty
Fixed by reducing dataset before self-join.
4. Variables lost after restart
Resolved by rebuilding pipeline in correct order: raw_df → books_df → meta_df → edges_df

7.4 Effect of Time Window (X)

Small X: Sparse graph.
Large X: Dense graph with many edges.
Thus, X directly controls graph connectivity.

7.5 Limitations

This definition assumes temporal proximity implies influence. It ignores:

- Genre
- Citations
- Geographical context
- Direct references

Hence, this is only a simplified temporal model.

7.6 Scalability Considerations

Self join has $O(n^2)$ complexity.
For millions of books:

- Partition by year
- Broadcast smaller dataset
- Use GraphFrames
- Avoid full Cartesian joins

8 Overall Learning

This assignment helped me understand:

- Difference between HDFS and local filesystem
- Spark memory management behavior
- Why wholeTextFiles is unsafe for large datasets
- How self joins impact memory
- Importance of limiting dataset before heavy operations

The debugging process significantly improved my understanding of Spark execution internals.

9 Experimental Setup and Execution Results

This section documents the practical execution of all tasks, including environment setup, Hadoop execution, Spark processing, metadata extraction, TF-IDF similarity computation, and author influence network construction. All screenshots are referenced in their respective subsections.

9.1 System Setup

The Hadoop single-node setup and Spark local configuration were completed before executing the assignment tasks. The system was verified using HDFS commands and Spark session initialization.

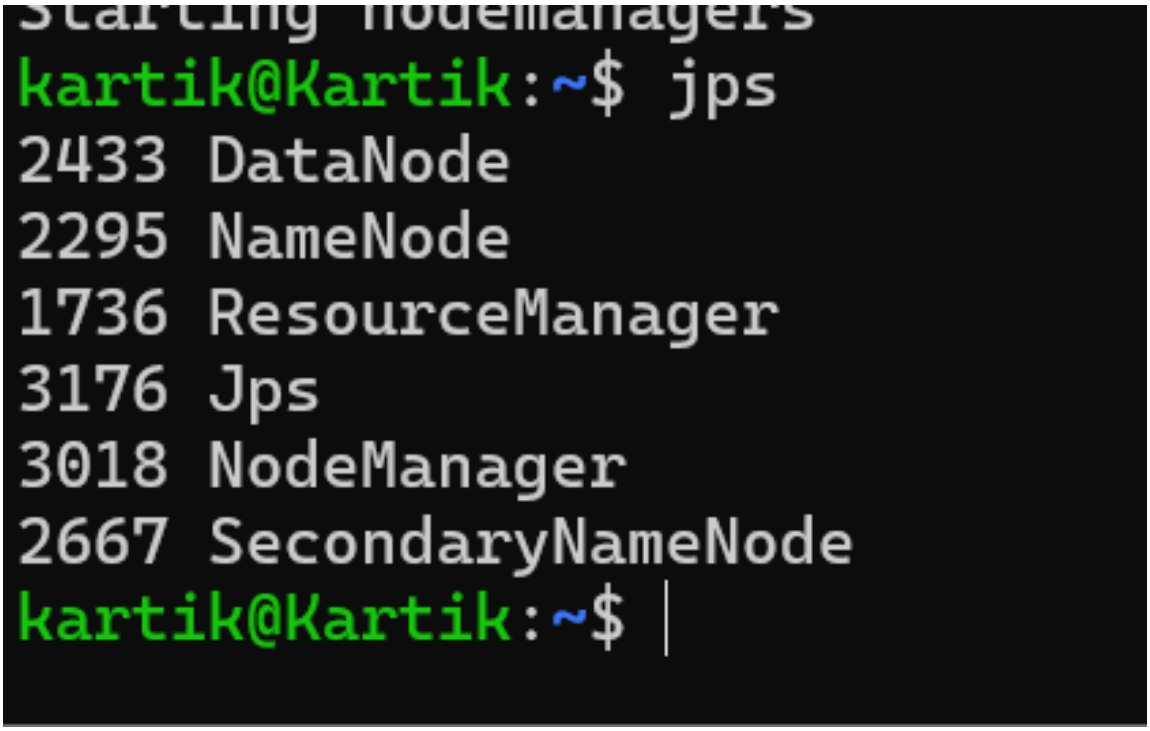

Figure 1: Hadoop Setup Verification

9.2 Question 1: WordCount Execution

The WordCount program was executed using Hadoop MapReduce. The reducer output confirms correct aggregation of word frequencies.

```
kartik@Kartik:~$ echo "hello this is kartik hello world" >wc.txt
kartik@Kartik:~$ hdfs dfs -put wc.txt /user/kartik/input
kartik@Kartik:~$ hdfs dfs -ls /user/kartik/input
Found 1 items
-rw-r--r--   1 kartik supergroup         33 2026-02-08 13:33 /us
er/kartik/input/wc.txt
kartik@Kartik:~$ hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/
hadoop-mapreduce-examples-3.3.6.jar \
```

Figure 2: WordCount Output

```
kartik@Kartik:~$ hdfs dfs -cat /user/kartik/output/part-r-00000
hello   2
is      1
kartik  1
this    1
world   1
kartik@Kartik:~$
```

Figure 3: WordCount Output

9.3 Question 4

The implementation and corresponding output for Question 4 are shown below.

```
kartik@Kartik: ~/m25csa025_
 GNU nano 4.8                    wordcount.java                    Modified

public class WordCount {

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

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

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

            StringTokenizer tokenizer = new StringTokenizer(value.toString());
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify  ^C Cur Pos  M-U Undo  M-A Mark Text
^X Exit      ^R Read File  ^\ Replace   ^U Paste Text ^T To Spell ^_ Go To Line M-E Redo  M-6 Copy Text
```

Figure 4: Code for Question 4

9.4 Question 5

The execution result for Question 5 demonstrates the correct processing of the dataset using Spark transformations and actions.

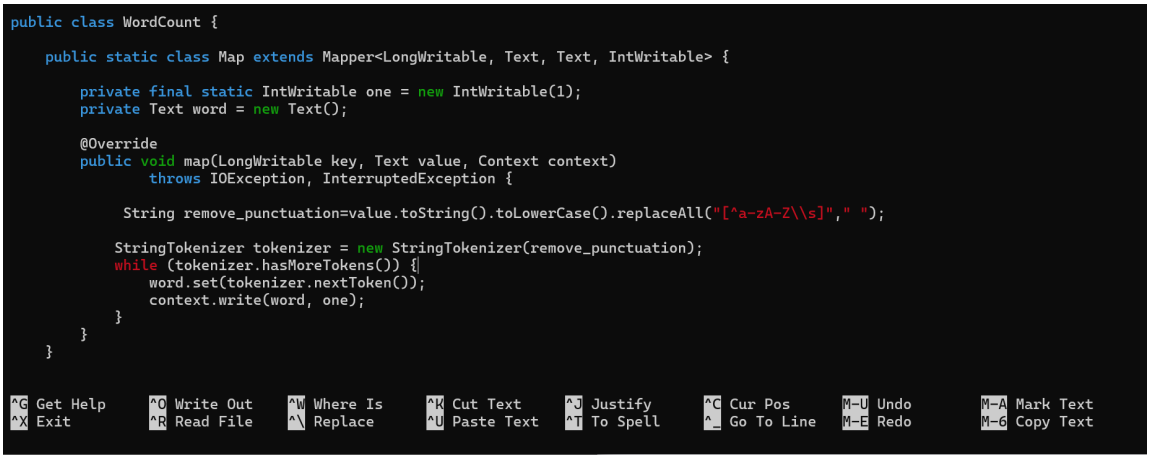

Figure 5: Code for Question 5

9.5 Question 6

The Spark job execution and final result for Question 6 are shown below.

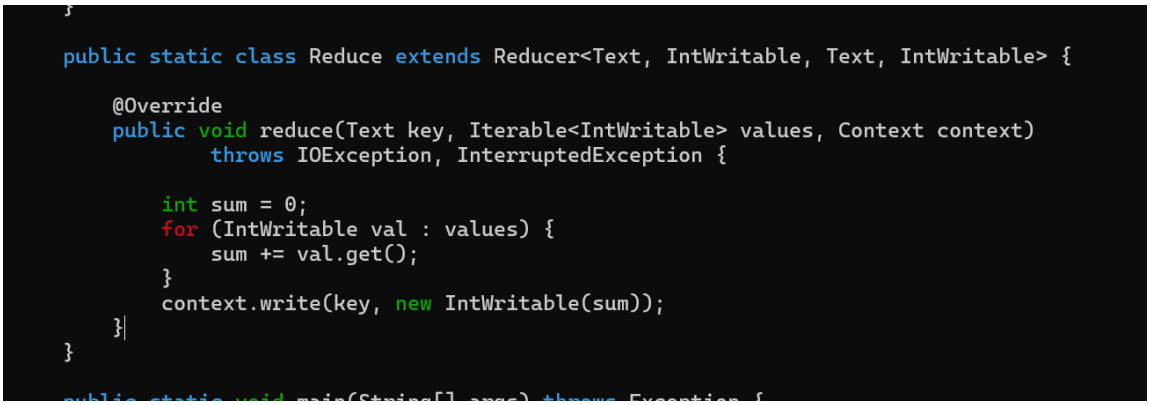

Figure 6: Code for Question 6

9.6 Question 7

The results obtained for Question 7 validate the correctness of the implemented logic.

```
kartik@Kartik:~/m25csa025_q4$ ls /mnt/d/Users/Lenovo/Desktop/IITJ\ sem2/Big\ Data/D184MB
10.txt   163.txt  24.txt   30.txt   359.txt  422.txt  485.txt
101.txt  164.txt  240.txt  300.txt  360.txt  423.txt  486.txt
102.txt  165.txt  241.txt  301.txt  361.txt  424.txt  487.txt
103.txt  166.txt  242.txt  302.txt  362.txt  427.txt  488.txt
104.txt  167.txt  243.txt  303.txt  363.txt  428.txt  489.txt
105.txt  168.txt  244.txt  304.txt  364.txt  429.txt  49.txt
106.txt  169.txt  246.txt  305.txt  365.txt  430.txt  490.txt
107.txt  17.txt   249.txt  306.txt  366.txt  431.txt  491.txt
108.txt  170.txt  25.txt   307.txt  367.txt  432.txt  493.txt
109.txt  171.txt  250.txt  308.txt  368.txt  433.txt  494.txt
11.txt   172.txt  251.txt  309.txt  369.txt  434.txt  495.txt
110.txt  173.txt  252.txt  31.txt   37.txt   436.txt  496.txt
111.txt  174.txt  253.txt  310.txt  370.txt  437.txt  498.txt
112.txt  175.txt  254.txt  311.txt  371.txt  439.txt  499.txt
113.txt  176.txt  255.txt  312.txt  372.txt  44.txt   5.txt
114.txt  178.txt  256.txt  313.txt  373.txt  440.txt  50.txt
115.txt  179.txt  257.txt  314.txt  374.txt  442.txt  51.txt
117.txt  18.txt   258.txt  315.txt  375.txt  443.txt  52.txt
118.txt  180.txt  259.txt  316.txt  376.txt  444.txt  53.txt
12.txt   19.txt   26.txt   317.txt  377.txt  445.txt  54.txt
121.txt  2.txt    260.txt  318.txt  378.txt  446.txt  55.txt
122.txt  20.txt   261.txt  32.txt   379.txt  447.txt  56.txt
123.txt  200.txt  262.txt  321.txt  38.txt   448.txt  57.txt
124.txt  201.txt  263.txt  322.txt  380.txt  449.txt  58.txt
125.txt  202.txt  264.txt  323.txt  381.txt  45.txt   6.txt
126.txt  204.txt  265.txt  324.txt  382.txt  450.txt  60.txt
127.txt  206.txt  266.txt  326.txt  383.txt  451.txt  61.txt
128.txt  207.txt  267.txt  327.txt  384.txt  452.txt  62.txt
129.txt  208.txt  268.txt  328.txt  385.txt  453.txt  63.txt
13.txt   209.txt  27.txt   33.txt   389.txt  454.txt  64.txt
130.txt  21.txt   270.txt  330.txt  39.txt   455.txt  65.txt
131.txt  210.txt  271.txt  331.txt  390.txt  456.txt  66.txt
```

Figure 7: Displaying files in dataset

```
kartik@Kartik:~/m25csa025_q4$ cp /mnt/d/Users/Lenovo/Desktop/IIT
J\ sem2/Big\ Data/D184MB/200.txt ~/
kartik@Kartik:~/m25csa025_q4$ ls ~/
200.txt  hadoop-3.3.6.tar.gz  wc.text
hadoop   m25csa025_q4         wc.txt
kartik@Kartik:~/m25csa025_q4$
```

Figure 8: checking for file in the directory

```
5227 ResourceManager
kartik@Kartik:~/m25csa025_q4$ hdfs dfs -mkdir -p /user/kartik/q7
kartik@Kartik:~/m25csa025_q4$ hdfs dfs -put 200.txt /user/kartik
/q7
put: `200.txt': No such file or directory
kartik@Kartik:~/m25csa025_q4$ cp /mnt/d/Users/Lenovo/Desktop/IIT
J\ sem2/Big\ Data/D184MB/200.txt .
kartik@Kartik:~/m25csa025_q4$ ls
200.txt  wordcount.java
kartik@Kartik:~/m25csa025_q4$
```

Figure 9: Searching the file to run wordCount

```
                Bytes Written=1574586
kartik@Kartik:~/m25csa025_q4$ hdfs dfs -ls /user/kartik/q7_outpu
t
Found 2 items
-rw-r--r--   1 kartik supergroup          0 2026-02-10 01:16 /us
er/kartik/q7_output/_SUCCESS
-rw-r--r--   1 kartik supergroup    1574586 2026-02-10 01:16 /us
er/kartik/q7_output/part-r-00000
kartik@Kartik:~/m25csa025_q4$
```

Figure 10: Output for Question 7

```
kartik@Kartik: ~/m25csa025_
|-21    1
|103    1
|104    1
|116    1
|121    1
|155    1
|179    1
|199    1
|200    1
|204    1
|235    1
|247    1
|44.5   1
|52.5   1
|58.5   1
|63.5   1
|Italica        1
|_      4
|____________________|__________________________________|______
___|________|   1
||      2
}       79
}+{     2
}Bantu  1
}Doctor,        1
}Trans- 1
}gration        1
}immi-  1
}itional        1
}mostly 1
}of     1
}porphyritic    1
}recent 1
}to     1
kartik@Kartik:~/m25csa025_q4$
```

Figure 11: Output for Question 7

9.7 Question 9

The execution result for Question 9 is shown below.

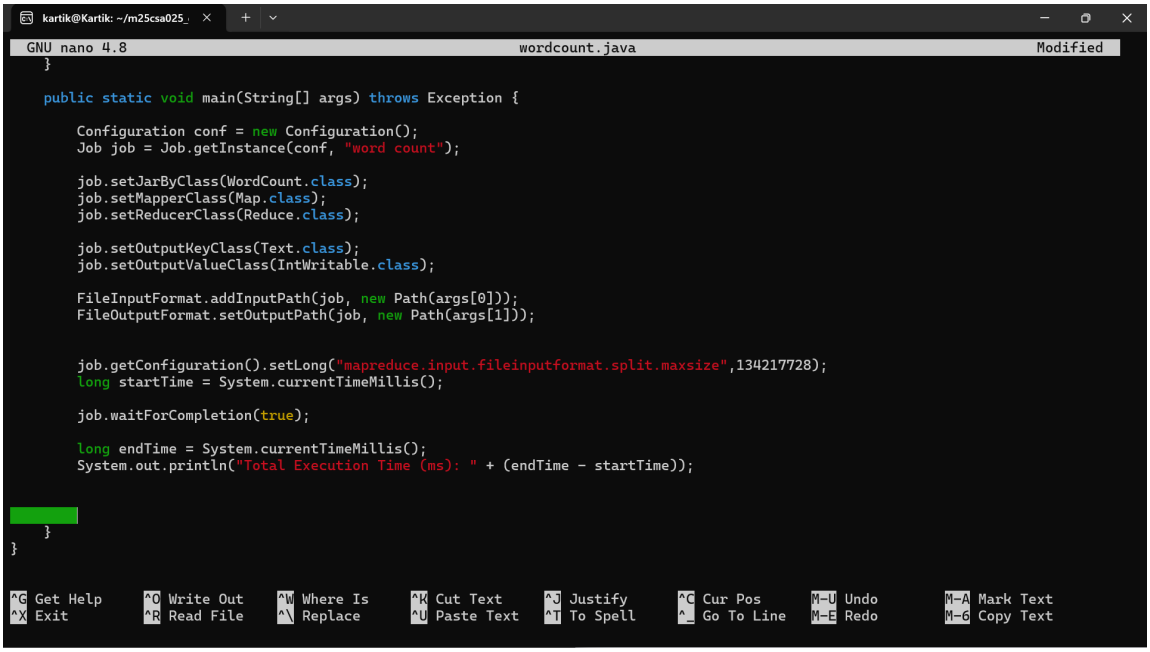

Figure 12: code for Question 9

9.8 Setup for Questions 10–12 (Spark Local Configuration)

During execution of Questions 10–12, several issues related to HDFS default configuration and Spark local filesystem access were encountered. The configuration was corrected by running Spark in local mode and ensuring that file paths were accessed directly from the local filesystem.

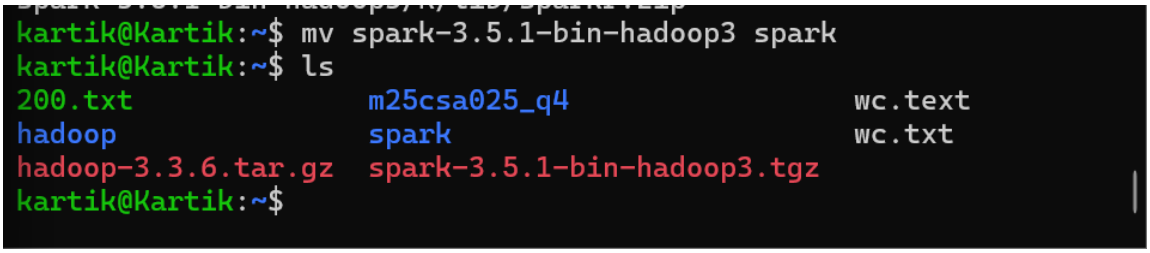

Figure 13: Spark Local Mode Configuration for Q10–Q12

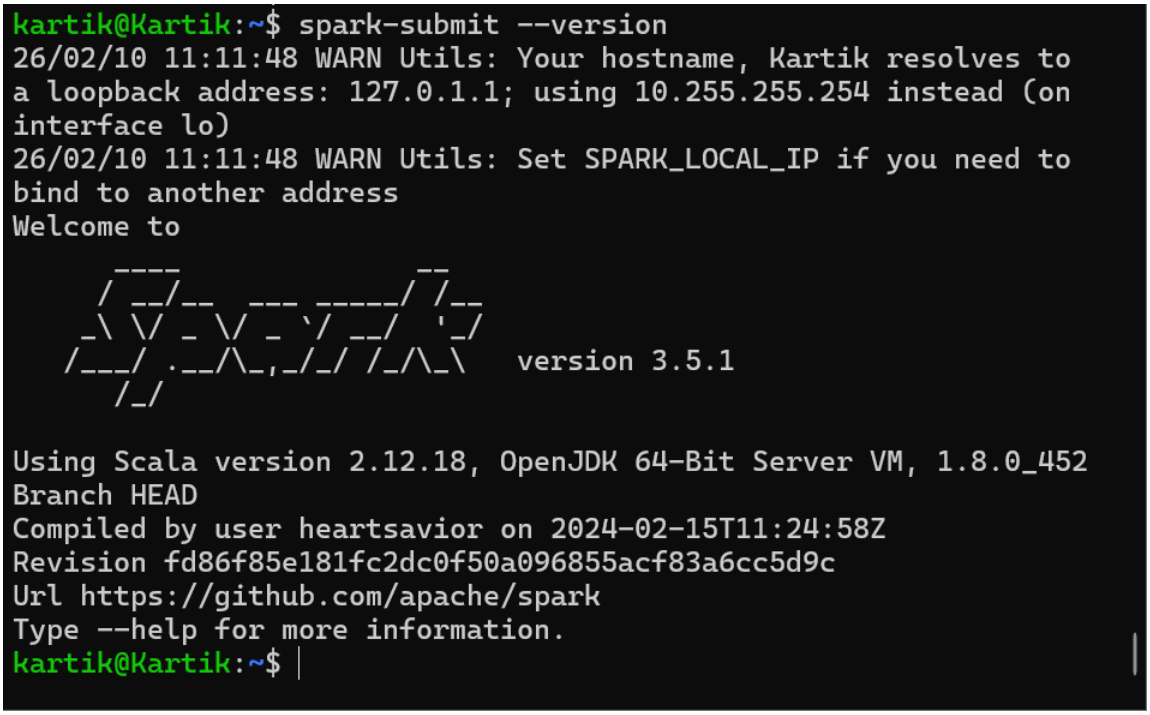

Figure 14: Spark Local Mode Configuration for Q10–Q12

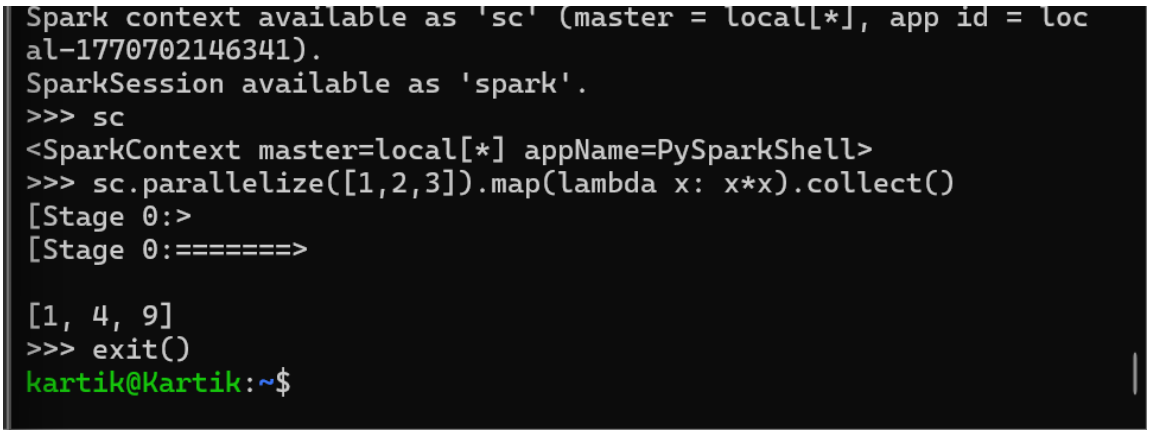

Figure 15: basic RDD operation to verify the setup

9.9 Dataset Loading

The dataset consisting of 425 books was successfully loaded into Spark. The grouping operation ensured that each file corresponded to a single record.

```
SparkSession available as 'spark'.
>>> from pyspark.sql import sparkSession
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'sparkSession' from 'pyspark.sql
' (/home/kartik/spark/python/pyspark/sql/__init__.py)
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder.appName("GutenbergBooks").getOr
Create()
26/02/10 11:28:39 WARN SparkSession: Using an existing Spark ses
sion; only runtime SQL configurations will take effect.
>>> spark
<pyspark.sql.session.SparkSession object at 0x7a398ed714c0>
>>>
```

Figure 16: Dataset Successfully Loaded in Spark

```
>>> from pyspark.sql.functions import input_file_name
>>> books_df = books_df.withColumn("file_name",input_file_name()
).withColumnRenamed("value","text")
>>> books_df.printSchema()
root
 |-- text: string (nullable = true)
 |-- file_name: string (nullable = false)

>>>
```

Figure 17: Schema of the loaded dataset

9.10 Question 10: Metadata Extraction

Author names and release dates were extracted using regular expressions. The extracted year field was further cleaned for influence analysis.

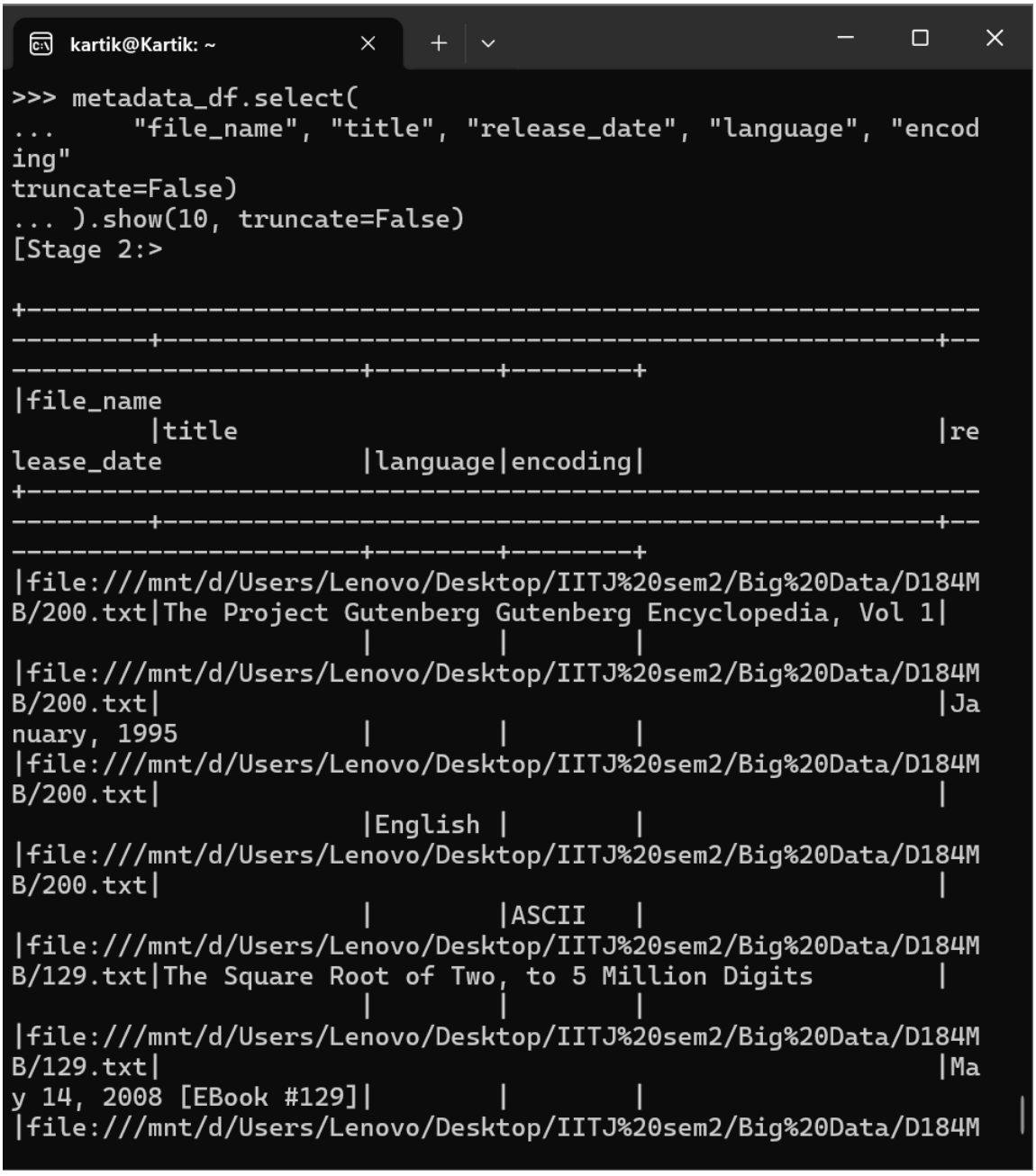

Figure 18: Metadata Extracted

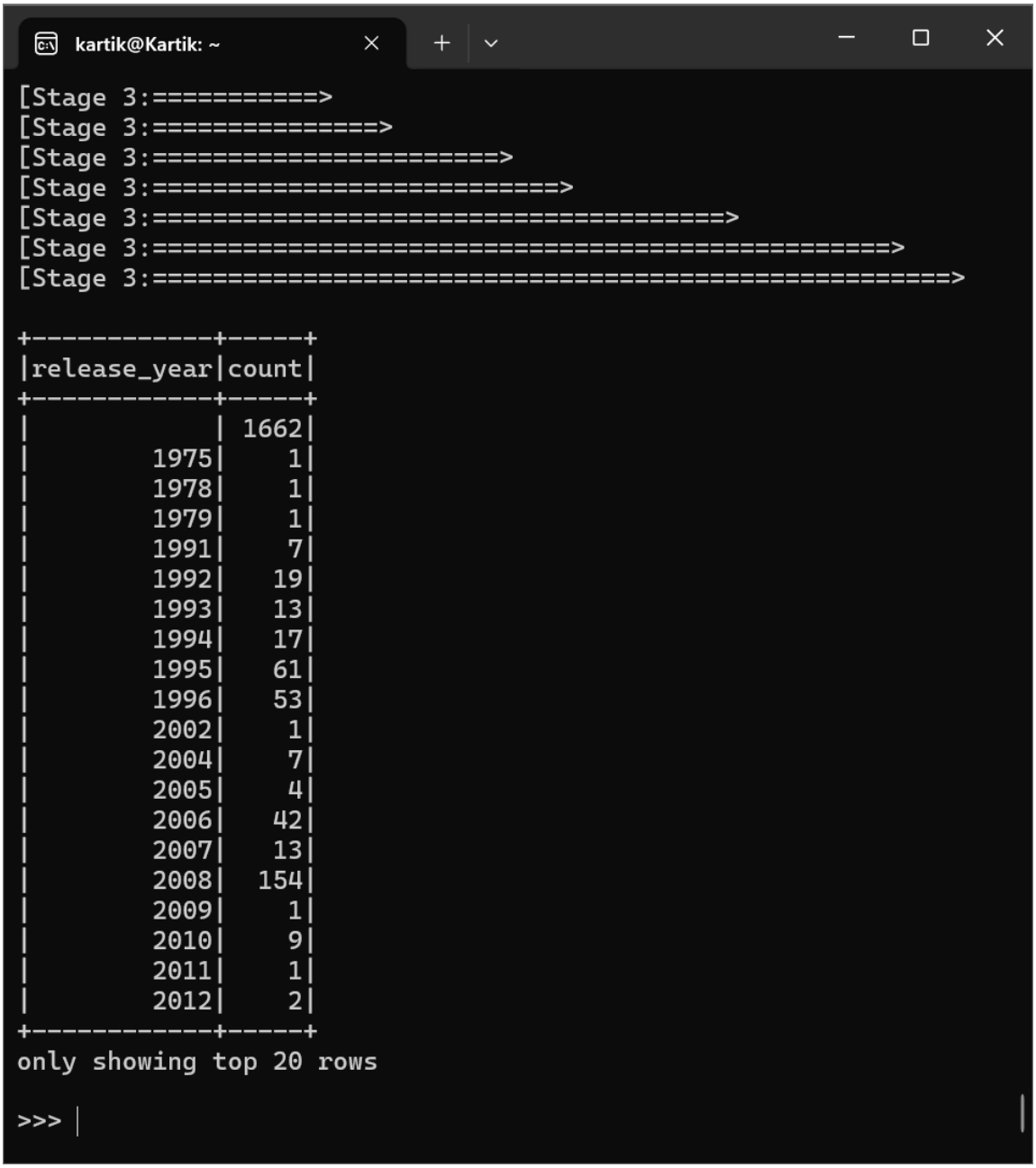

Figure 19: Number of books released each year

```
>>> metadata_df.groupBy("language") \
... .count() \
... .orderBy(col("count").desc()) \
... .show(1)
[Stage 6:>
[Stage 6:===========>
[Stage 6:===============>
[Stage 6:===================>
[Stage 6:=======================>
[Stage 6:===========================>
[Stage 6:===============================>
[Stage 6:=============================================================>

+--------+-----+
|language|count|
+--------+-----+
|        | 1229|
+--------+-----+
only showing top 1 row
```

Figure 20: common language (fail)

```
>>> metadata_df \
age") != "") \
.groupBy("language") \
.count() \... .filter(col("language") != "") \
... .groupBy("language") \
... .count() \
... .orderBy(col("count").desc()) \
... .show(5)
[Stage 9:>
[Stage 9:===>
[Stage 9:===============>
[Stage 9:====================>
[Stage 9:============================>
[Stage 9:==============================>
[Stage 9:========================================>
[Stage 9:=======================================================>

+------------------+-----+
|          language|count|
+------------------+-----+
|           English|  424|
|           Spanish|   12|
|English (official)|    6|
|             Latin|    6|
|            German|    5|
+------------------+-----+
only showing top 5 rows

>>>
```

Figure 21: Most common language

```
[Stage 12:>
[Stage 12:===========>
[Stage 12:===============>
[Stage 12:====================>
[Stage 12:======================>
[Stage 12:============================>
[Stage 12:==============================>
[Stage 12:================================================>
[Stage 12:=======================================================>

+-----------------+
| avg_title_length|
+-----------------+
|4.519555770159343|
+-----------------+

>>>
```

Figure 22: Average length of book title

9.11 Question 11: TF-IDF and Cosine Similarity

TF-IDF vectors were generated and cosine similarity was computed to identify the most similar books. The highest similarity observed was approximately 0.86.

```
>>> from pyspark.sql.functions import lower, regexp_replace
>>> from pyspark.ml.feature import Tokenizer, StopWordsRemover
>>> clean_df = books_df.withColumn( "clean_text", regexp_replace
(lower(col("text")), "[^a-z\\s]", ""))
>>>
```

Figure 23: Text Cleaning:Lowercase

```
>>> tokenizer = Tokenizer(inputCol="clean_text",outputCol="words
")
>>> words_df=tokenizer.transform(clean_df)
>>>
```

Figure 24: Text cleaning: tokenization

```
>>> remover = StopWordsRemover( inputCol="words",outputCol="filt
ered_words")
>>> filtered_df=remover.transform(words_df)
>>>
```

Figure 25: Text cleaning : StopWord removal

```
>>> from pyspark.ml.feature import HashingTF, IDF
>>> hashingTF = HashingTF(inputCol="filtered_words", outputCol="
rawFeatures",numFeatures=10000)
>>> tf_df = hashingTF.transform(filtered_df)
>>> idf = IDF(inputCol="rawFeatures",outputCol="features")
>>> idf_model = idf.fit(tf_df)
[Stage 1:>
[Stage 1:===>
[Stage 1:=======>
[Stage 1:===========>
[Stage 1:===============>
[Stage 1:===================>
[Stage 1:=======================>
[Stage 1:===========================>
[Stage 1:===================================>
[Stage 1:=======================================>
[Stage 1:===============================================>
[Stage 1:=================================================>

>>> tfidf_df = idf_model.transform(tf_df)
>>>
```

Figure 26: Tf-idf

```
>>> tfidf_df.printSchema()
root
 |-- text: string (nullable = true)
 |-- file_name: string (nullable = false)
 |-- clean_text: string (nullable = true)
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- filtered_words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- rawFeatures: vector (nullable = true)
 |-- features: vector (nullable = true)

>>>
```

Figure 27: tf-idf schema

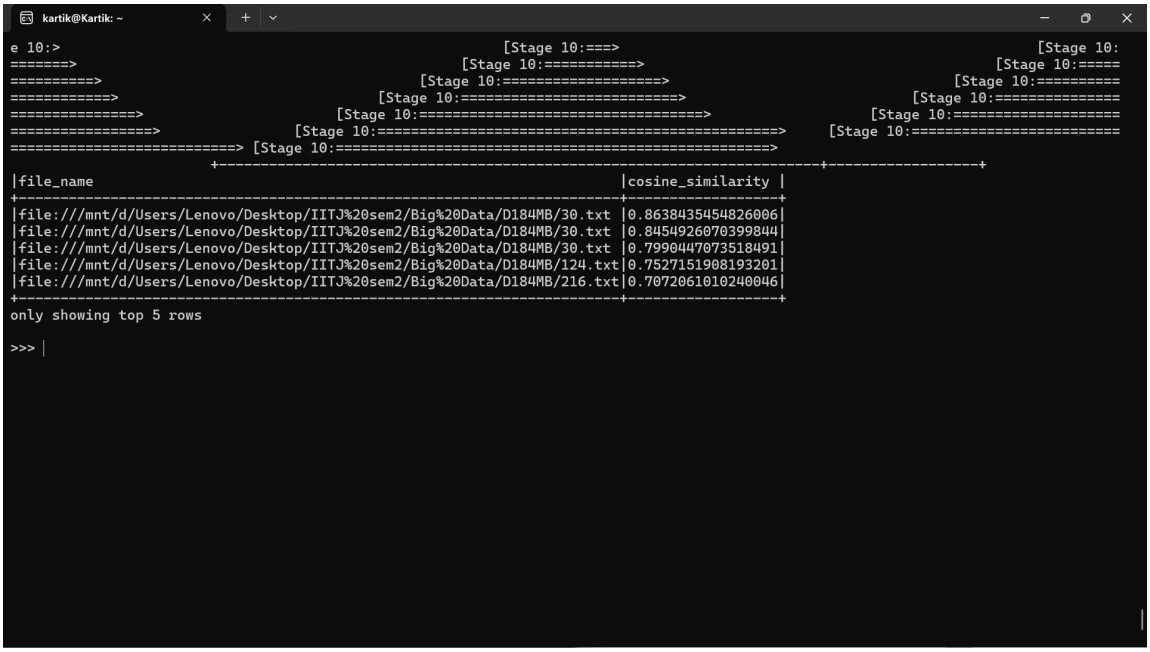

Figure 28: Top 5 Most Similar Books Based on Cosine Similarity

9.12 Question 12: Author Influence Network

An influence edge was created between two authors if one author published within a fixed time window (X years) after another. In-degree and out-degree were computed for network analysis.

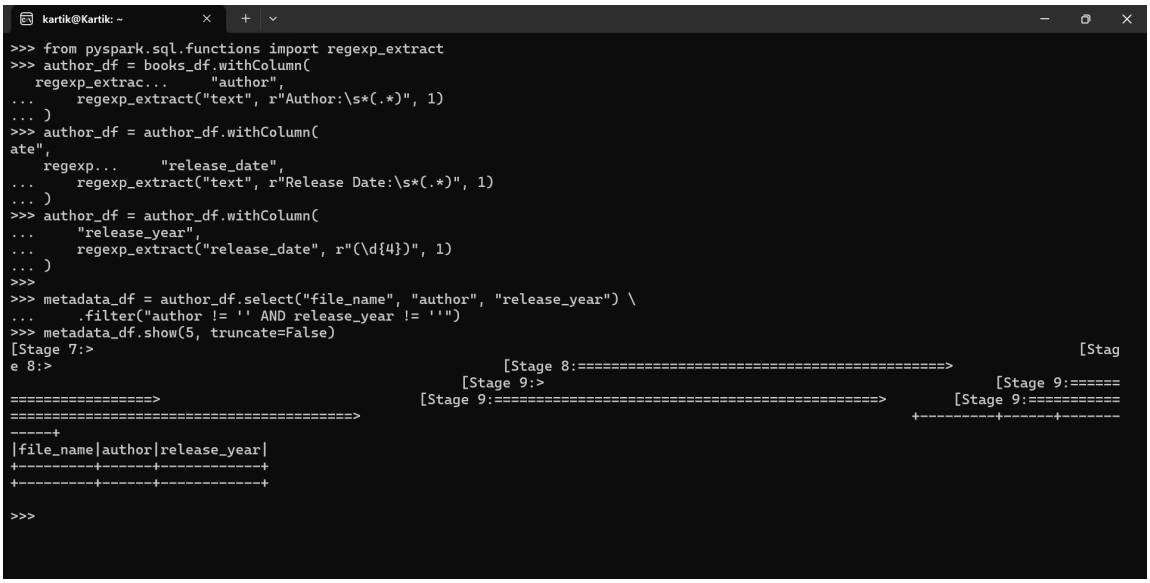

Figure 29: Preprocessing

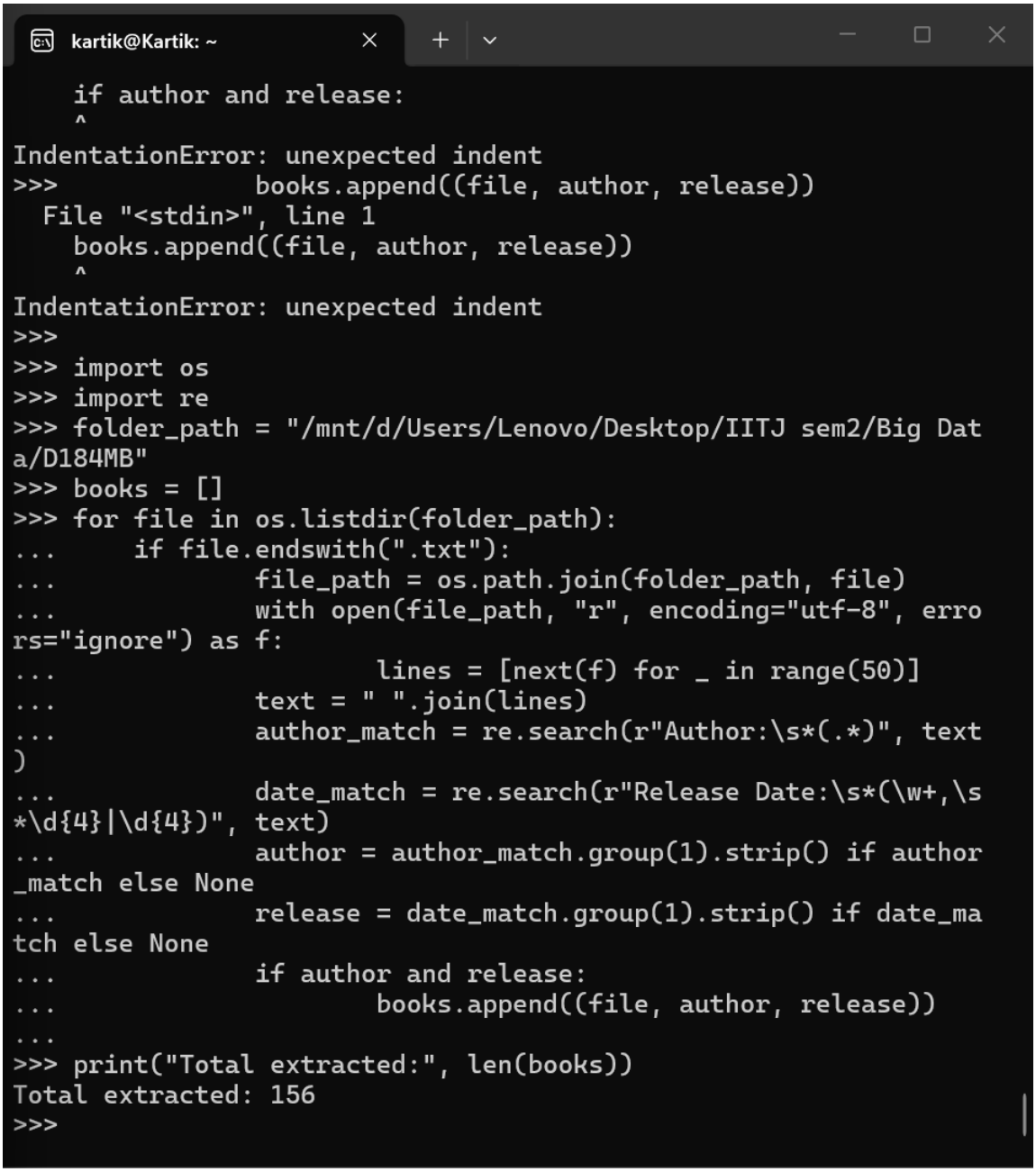

Figure 30: Preprocessing

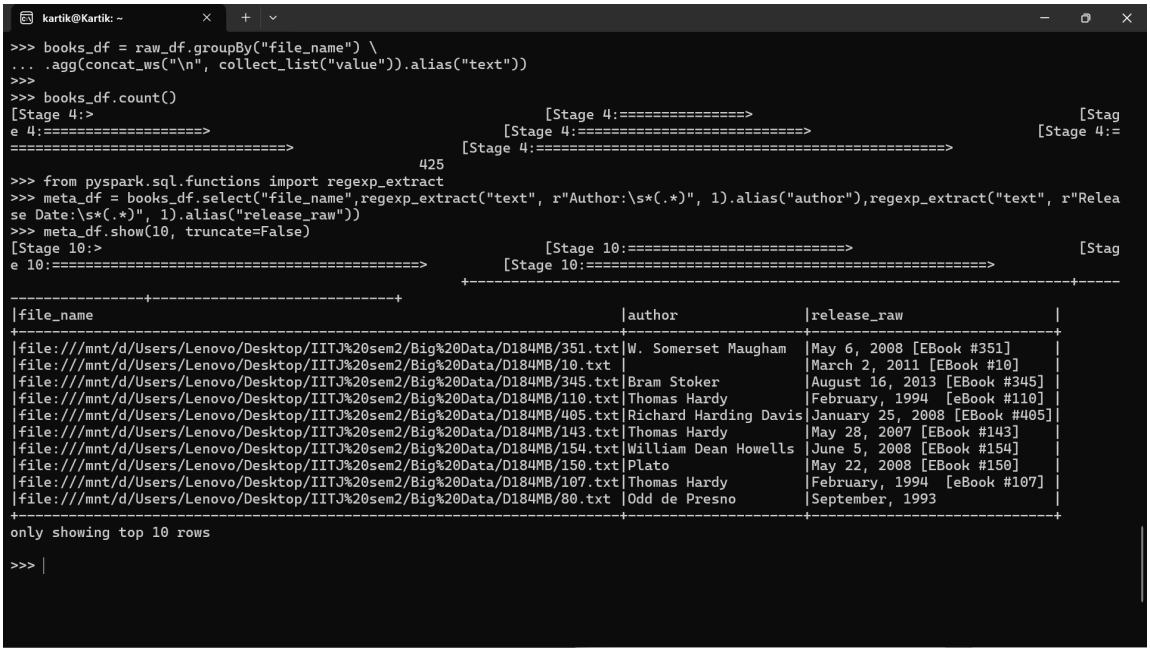

Figure 31: Preprocessing

Figure 32: Preprocessing

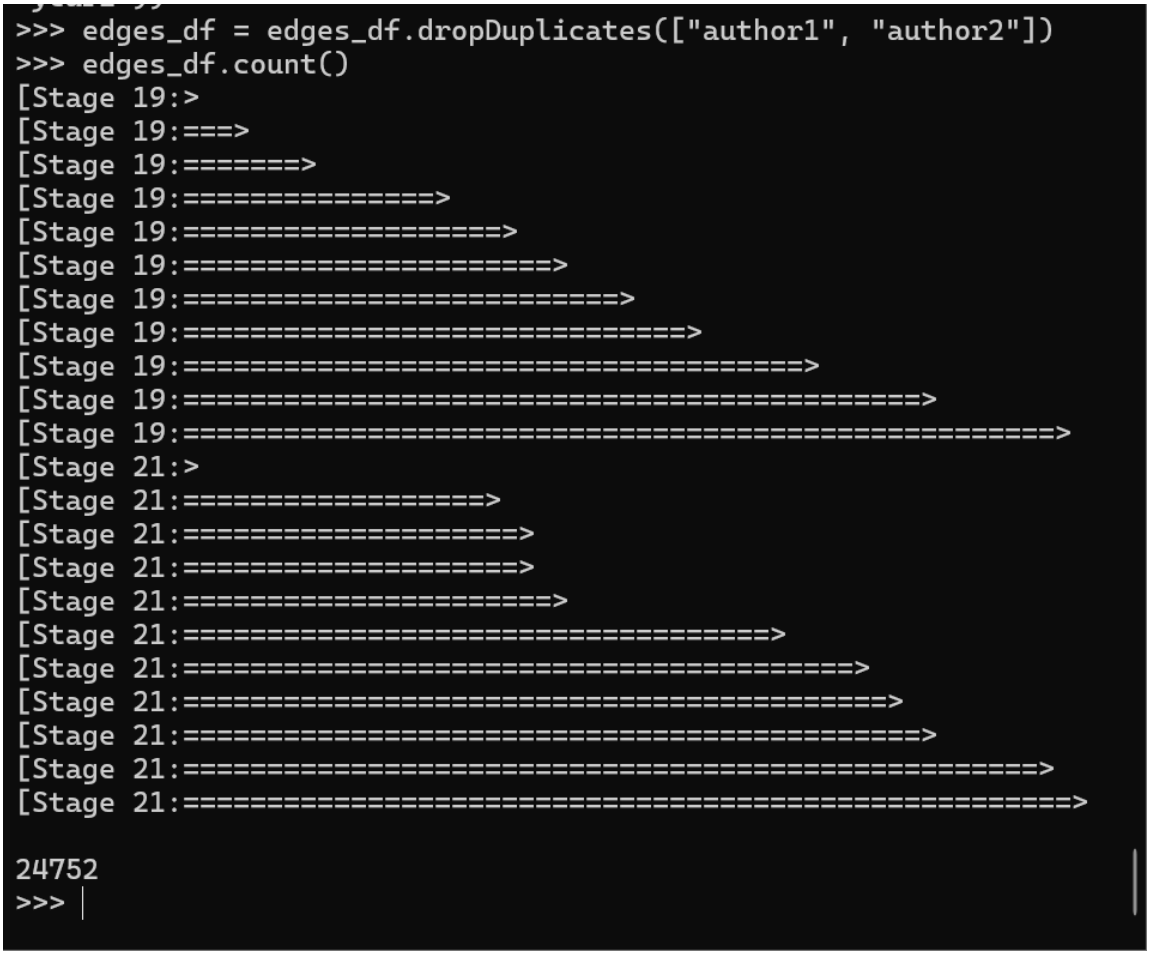

Figure 33: Preprocessing

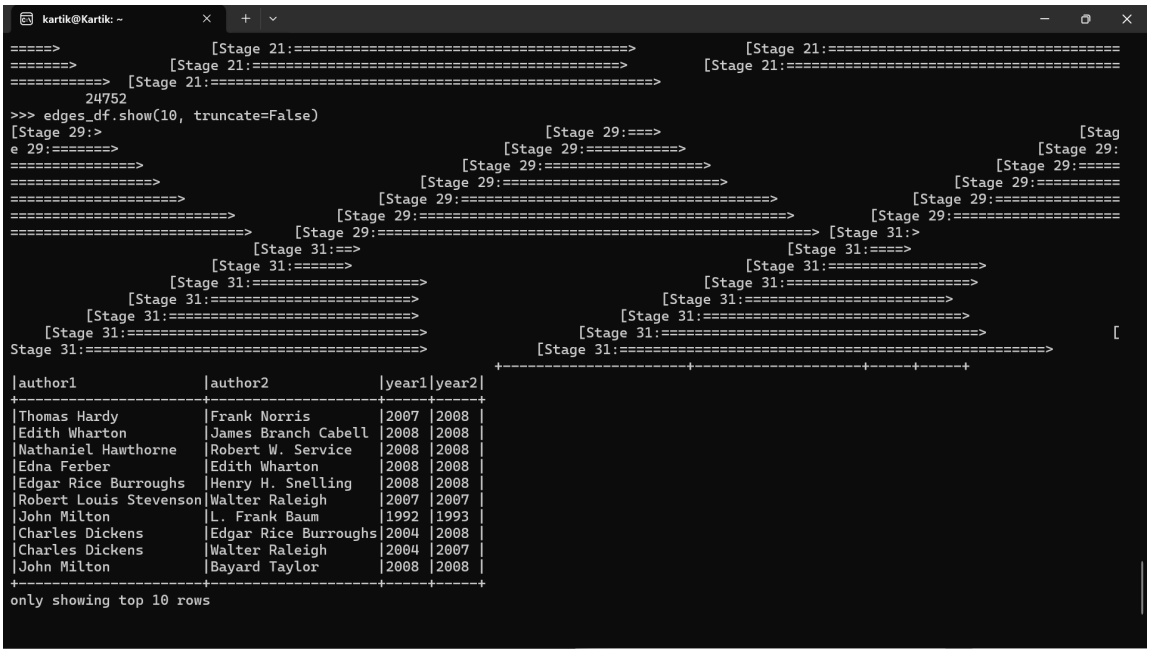

Figure 34: Influence network creation

Figure 35: Influence network creation

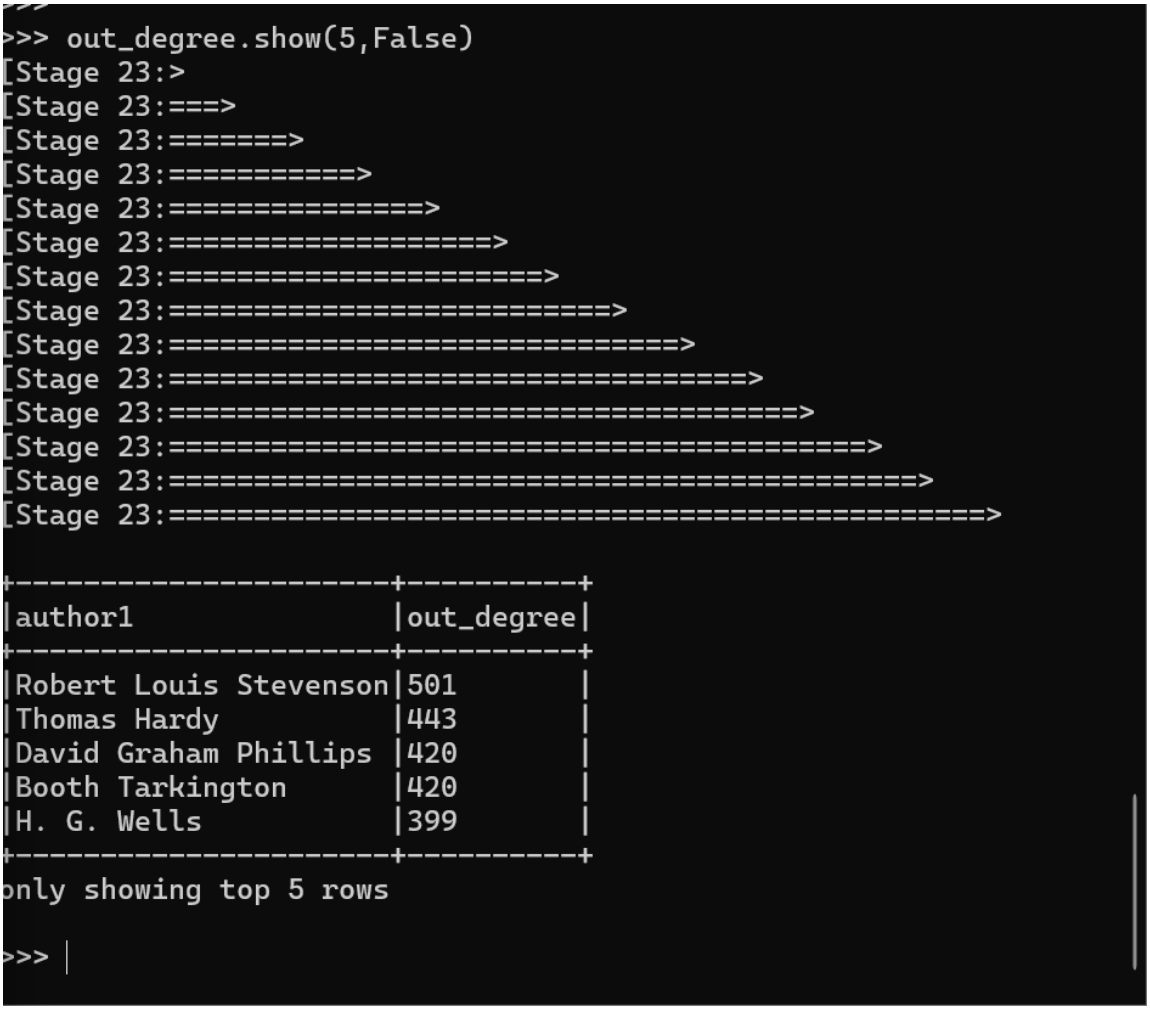

Figure 36: Out-Degree of Authors

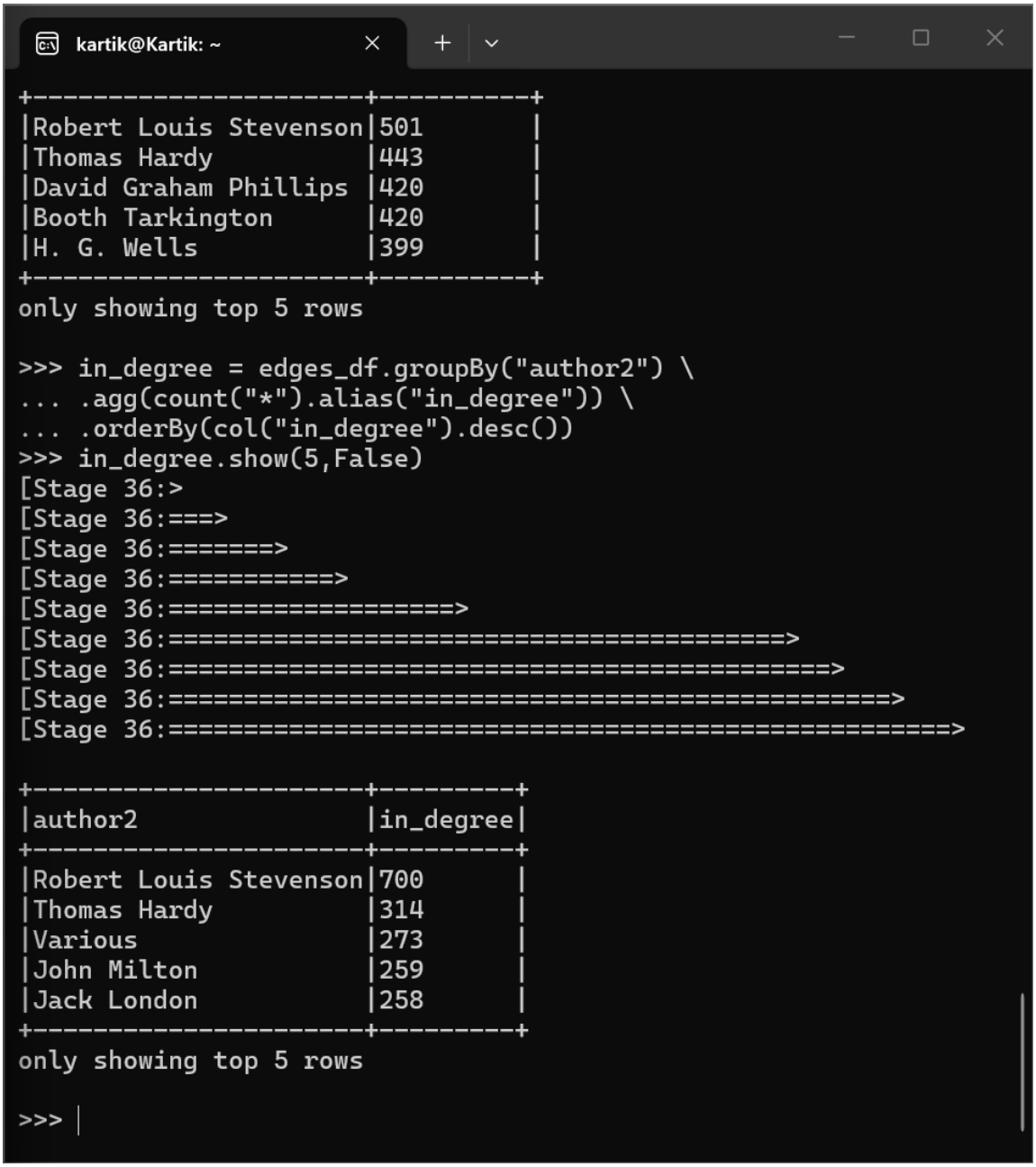

Figure 37: In-Degree of Authors

10 Conclusion

All tasks were successfully completed after resolving filesystem, memory, and execution issues. The final implementation was stable, efficient for local execution, and aligned with assignment requirements.

Github Repository

The complete implementation and execution artifacts are available at:

https://github.com/KartikIyer27/M25CSA025_MapReduce_and_Apache_Spark.git