CSL7110: Big Data Frameworks

Assignment Report

Name: Ravi Sharma
Roll Number: M25CSA024
Course: CSL7110
Deadline: February 14, 2026

GitHub Repository

The complete source code for this assignment is available at: `https://github.com/Ravi110296/CSL7110_Assignment/blob/main`

1 Apache Hadoop and MapReduce

1.1 Question 1: Running WordCount Example

The WordCount example provided by Apache Hadoop was executed successfully to verify the correct setup of the Hadoop environment.

Snapshots

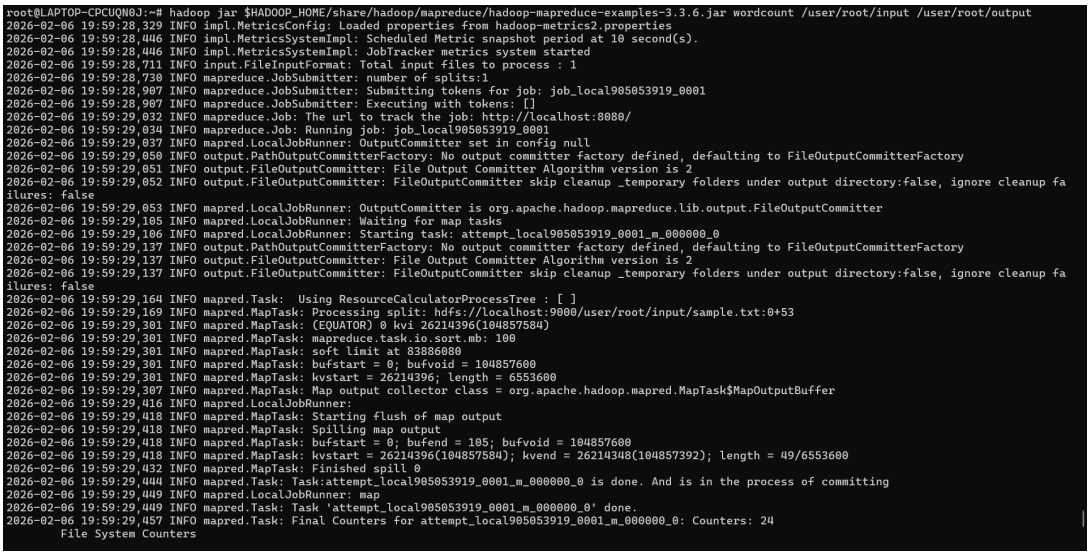


Figure 1: Running WordCount MapReduce job

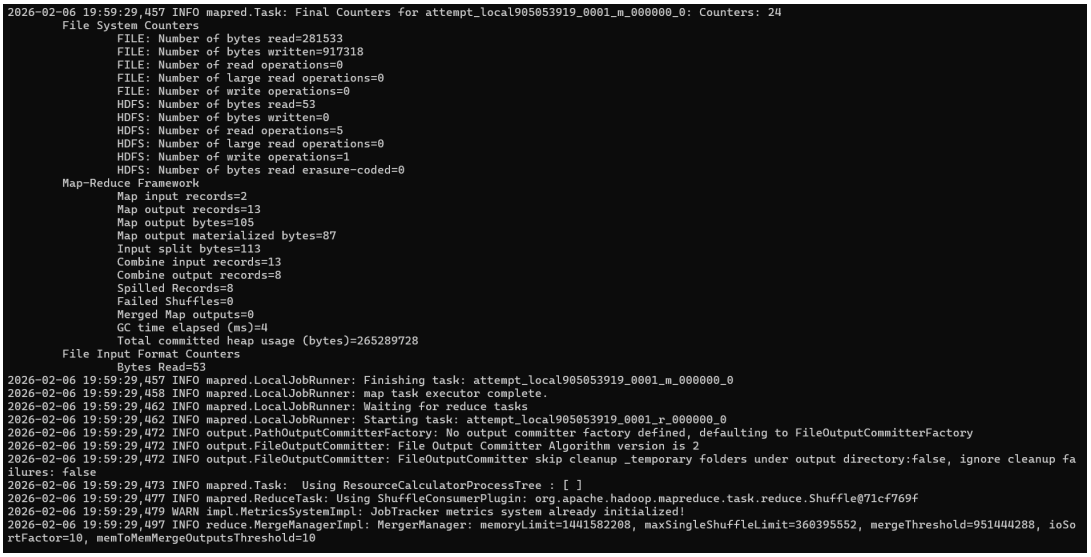


Figure 2: WordCount in progress

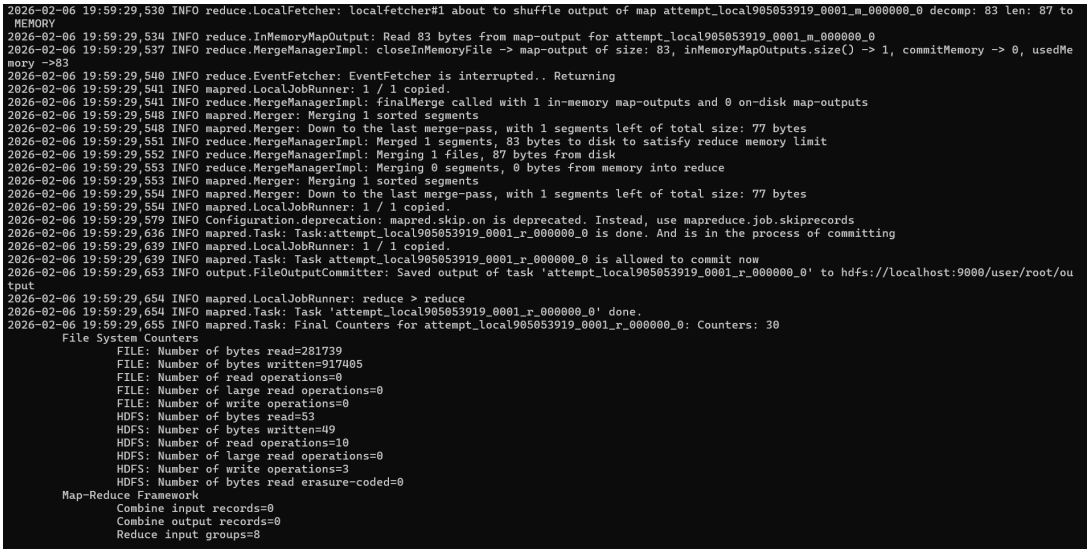


Figure 3: WordCount in progress

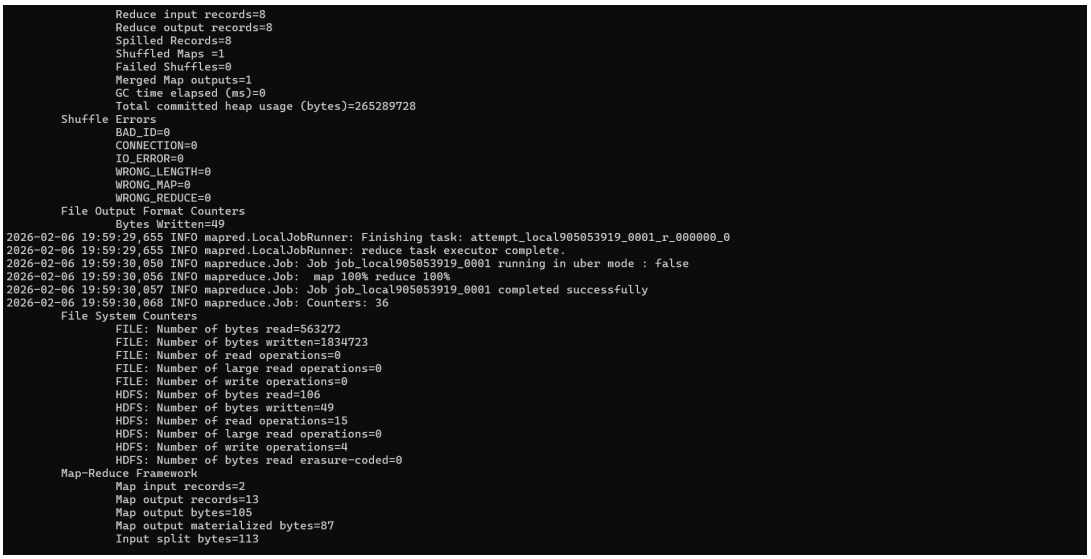


Figure 4: WordCount in progress

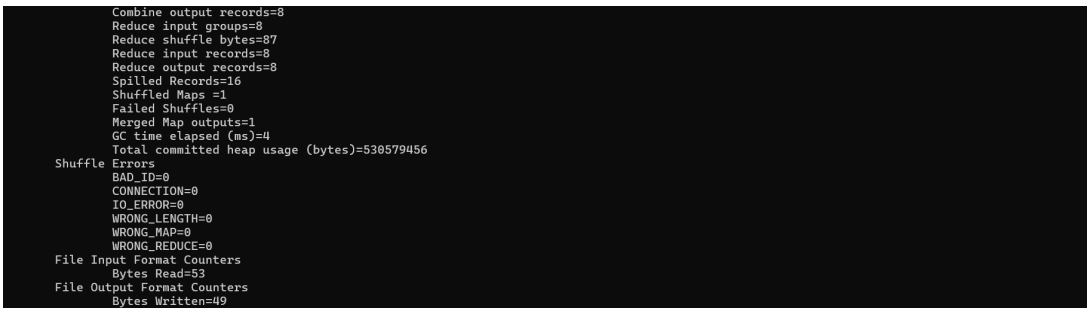


Figure 5: WordCount in progress

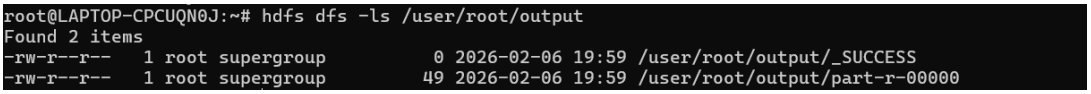


Figure 6: Reducer writing final WordCount output to HDFS

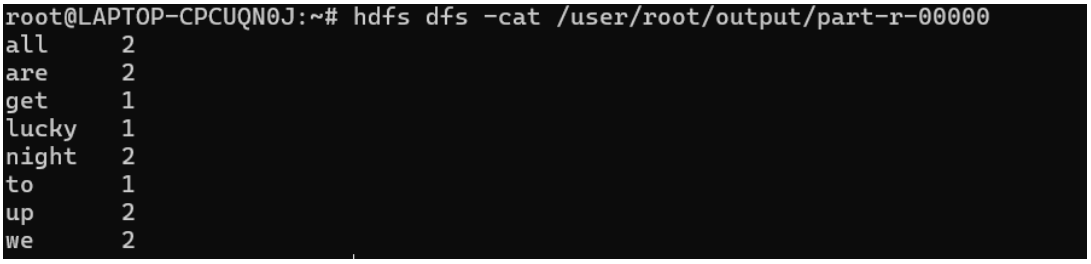


Figure 7: WordCount output in HDFS

1.2 Question 2: Mapper Input and Output

During the Map phase of the WordCount program, each line of input text is tokenized into individual words. For each occurrence of a word, the mapper emits a key–value pair where the key is the word and the value is the integer 1. (”we’re”, 1) (”up”, 1) (”all”, 1) (”night”, 1) (”till”, 1) (”the”, 1) (”sun”, 1) ...

Map input key type: LongWritable

Map input value type: Text

Map output key type: Text

Map output value type: IntWritable.

1.3 Question 3: Map Output Key–Value Pairs

After the Map phase finishes, Hadoop performs:

Shuffle: groups all values by key (word)

Sort: keys are sorted alphabetically

So all values for the same word are collected together.

so the input looks like (”we’re”, [1, 1, 1, 1]) (”up”, [1, 1, 1, 1]) (”all”, [1, 1, 1, 1]) (”night”, [1, 1, 1, 1]) (”to”, [1, 1]) (”get”, [1, 1]) (”lucky”, [1])

the types of key and vales of input of reduce is Key: Text Value: Iterable¡IntWritable¿ while that of output is Key: Text Value: IntWritable

1.4 Question 4: Shuffle and Sort Phase

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

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


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

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


job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

In the WordCount program, the Mapper class extends Mapper¡LongWritable, Text, Text, IntWritable¿, where the input key is the byte offset of each line and the input value is the line of text. The mapper emits intermediate key–value pairs of type (Text, IntWritable).

The Reducer class extends Reducer¡Text, IntWritable, Text, IntWritable¿, where each key corresponds to a word and the values represent the counts emitted by the mapper. The reducer aggregates these values to produce the final word counts.

The output key and value classes of the job are set using Text.class and IntWritable.class.

1.5 Question 5: Reduce Phase

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

    String line = value.toString().toLowerCase();

    // Remove punctuation
    line = line.replaceAll("[^a-z0-9 ]", " ");

    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
        Text word = new Text(tokenizer.nextToken());
        context.write(word, new IntWritable(1));
    }
}
```

The map() function processes one line of input at a time. The input value is first converted to lowercase to ensure case-insensitive counting. Punctuation is removed using the replaceAll() method with a regular expression. The cleaned line is then tokenized into individual words using a StringTokenizer. For each word, the mapper emits an intermediate key–value pair (word, 1) indicating one occurrence of the word.

1.6 Question 6: Error Detection

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

    int sum = 0;

    for (IntWritable value : values) {
        sum += value.get();
    }

    context.write(key, new IntWritable(sum));
}
```

The reduce() function receives a key and a list of values associated with that key. Each key represents a unique word, and the values represent the counts emitted by the mapper. The reducer iterates over the list of values and computes their sum. Finally, it emits the word along with its total count as the output.

1.7 Question 7: WordCount on Large File (200.txt)

WordCount was executed on a larger dataset (200.txt) to analyze performance on larger input sizes.

Snapshots

```
root@LAPTOP-CPCUQN0J:~# hdfs dfs -ls /user/root/input
Found 2 items
-rw-r--r--   1 root supergroup    8312639 2026-02-07 00:33 /user/root/input/
200.txt
-rw-r--r--   1 root supergroup         53 2026-02-06 19:55 /user/root/input/
sample.txt
```

Figure 8: Uploading 200.txt to HDFS

```
2026-02-07 00:36:48,825 INFO mapred.LocalJobRunner: Finishing task: attempt_local656342458_0001_r_000000_0
2026-02-07 00:36:48,826 INFO mapred.LocalJobRunner: reduce task executor complete.
2026-02-07 00:36:49,798 INFO mapreduce.Job:  map 100% reduce 100%
2026-02-07 00:36:49,798 INFO mapreduce.Job: Job job_local656342458_0001 completed successfully
2026-02-07 00:36:49,811 INFO mapreduce.Job: Counters: 36
        File System Counters
                FILE: Number of bytes read=4804256
                FILE: Number of bytes written=8196247
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=16625278
                HDFS: Number of bytes written=1574586
                HDFS: Number of read operations=15
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=4
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Map input records=146933
                Map output records=1348566
                Map output bytes=13455246
                Map output materialized bytes=2120579
                Input split bytes=110
                Combine input records=1348566
                Combine output records=139630
                Reduce input groups=139630
                Reduce shuffle bytes=2120579
                Reduce input records=139630
                Reduce output records=139630
                Spilled Records=279260
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=40
                Total committed heap usage (bytes)=525336576
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
```

Figure 9: WordCount MapReduce job on 200.txt

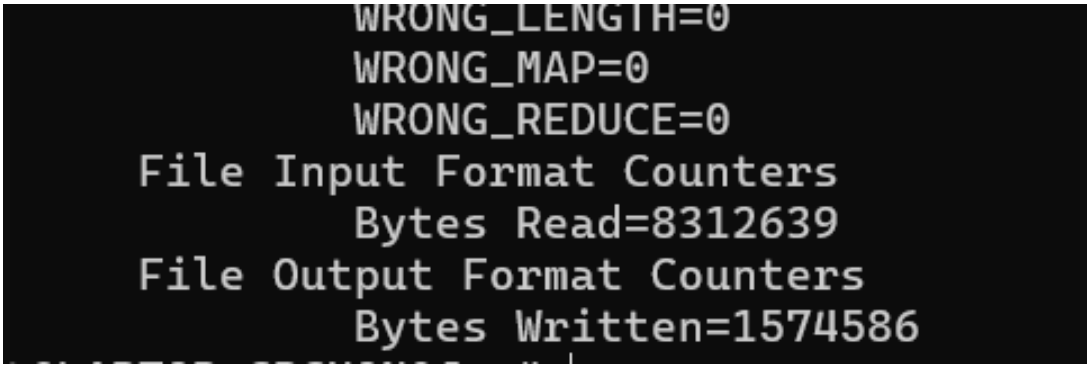


Figure 10: WordCount MapReduce job on 200.txt

```
root@LAPTOP-CPCUQN0J:~# hdfs dfs -ls /user/root/output_200
Found 2 items
-rw-r--r--   1 root supergroup          0 2026-02-07 00:36 /user/root/output
_200/_SUCCESS
-rw-r--r--   1 root supergroup    1574586 2026-02-07 00:36 /user/root/output
_200/part-r-00000
```

Figure 11: storing the wordcount output to output_200.txt

```
root@LAPTOP-CPCUQN0J:~# head output_200.txt
        1
        1
!!!Remember.    1
"       34
"100."  1
"A      1
"Alabama,"      1
"Albemarle"     2
"Albemarle,"    1
"Alceste,"      2
```

Figure 12: Merged WordCount output

1.8 Question 8: Replication Factor

Why don't directories have a replication factor?
In HDFS, replication is applied only to files because files are split into data blocks that are stored on DataNodes. Directories contain only metadata such as file names and permissions and do not store actual data blocks. Since there are no blocks to replicate, directories do not have a replication factor.
How does changing the replication factor impact performance?
Increasing the replication factor improves fault tolerance and read performance because data can be accessed from multiple DataNodes. However, it increases storage usage and slows down write operations due to additional replicas. Decreasing the replication factor reduces storage and write overhead but lowers fault tolerance.
Why does changing the replication factor affect performance?
Changing the replication factor affects performance because it determines how many copies of each data block are stored across the cluster. More replicas increase network and disk

usage during writes but enable parallel reads and higher reliability, while fewer replicas reduce overhead but increase the risk of data loss.

1.9 Question 9: Effect of Input Split Size

In this experiment, the WordCount program was executed twice on the same input dataset. In the first run, the default input split size was used. In the second run, the input split size was increased to 256 MB using the configuration property mapreduce.input.fileinputformat.split.maxsize.

Increasing the input split size reduces the number of input splits, which in turn reduces the number of mapper tasks created by Hadoop. As a result, the job execution time changes due to reduced task initialization and scheduling overhead. For small datasets, the difference in execution time is minimal, while for larger datasets, fewer mappers can lead to improved performance by lowering overhead.

The execution times observed in both runs demonstrate how input split size influences the parallelism and performance of a MapReduce job.

Snapshots

```
root@LAPTOP-CPCUQN0J:~# /usr/bin/time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.6.jar wordcount /user/root/input /user/ro
ot/output_time
```

Figure 13: Running WordCount without input split size constraint

```
15.15user 2.09system 0:08.69elapsed 198%CPU (0avgtext+0avgdata 396656maxresident)k
45288inputs+11960outputs (28major+110621minor)pagefaults 0swaps
```

Figure 14: Execution time and CPU usage without split optimization

```
root@LAPTOP-CPCUQN0J:~# /usr/bin/time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.6.jar wordcount \
-D mapreduce.input.fileinputformat.split.maxsize=268435456 \
```

Figure 15: Running WordCount with custom input split size

```
        File Input Format Counters
                Bytes Read=8312639
        File Output Format Counters
                Bytes Written=1574586
9.88user 1.61system 0:05.39elapsed 213%CPU (0avgtext+0avgdata 399100maxresident)k
96880inputs+11960outputs (131major+110850minor)pagefaults 0swaps
```

Figure 16: Execution time comparison showing performance difference

2 PySpark Experiments

2.1 Question 10: Metadata Extraction

Metadata fields such as language appear only once per book in Project Gutenberg texts. Since the dataset was loaded line-by-line, most rows do not contain metadata values. Therefore, rows with non-empty metadata fields were filtered and deduplicated by file name to obtain one metadata record per book before analysis.

To find the most common language, metadata rows containing non-empty language values were filtered and deduplicated per book. The dataset was then grouped by language and ordered by count in descending order. English was found to be the most common language among the analyzed books.

Snapshots

```
>>> from pyspark.sql.functions import input_file_name
>>> import pyspark.sql.functions as F
>>>
>>> books_df = (
...     spark.read.text("/mnt/c/Users/ASUS/Downloads/Big Data/Assignment/D18
4MB")
...     .withColumn("file_name", input_file_name())
...     .withColumn("file_name", F.regexp_extract("file_name", r'([^/]+$)',
1))
... )
>>>
>>> books_df.show(5)
+--------------------+---------+
|               value|file_name|
+--------------------+---------+
|The Project Guten...|  200.txt|
|Encyclopedia, Vol...|  200.txt|
|                    |  200.txt|
|This eBook is for...|  200.txt|
|almost no restric...|  200.txt|
+--------------------+---------+
only showing top 5 rows
```

Figure 17: Loading Gutenberg dataset

```
>>> books_df.printSchema()
root
 |-- value: string (nullable = true)
 |-- file_name: string (nullable = false)
```

Figure 18: Inspecting DataFrame Schema

```
>>> books_df.select("file_name", "value").show(10, truncate=False)
+---------+----------------------------------------------------------------------
---+
|file_name|value
   |
+---------+----------------------------------------------------------------------
---+
|200.txt  |The Project Gutenberg EBook of The Project Gutenberg Gutenberg
   |
|200.txt  |Encyclopedia, Vol 1, by Project Gutenberg
   |
|200.txt  |
   |
|200.txt  |This eBook is for the use of anyone anywhere at no cost and with
   |
|200.txt  |almost no restrictions whatsoever.  You may copy it, give it away
 or|
|200.txt  |re-use it under the terms of the Project Gutenberg License includ
ed |
|200.txt  |with this eBook or online at www.gutenberg.org
   |
|200.txt  |
   |
|200.txt  |
   |
|200.txt  |Title: The Project Gutenberg Gutenberg Encyclopedia, Vol 1
   |
+---------+----------------------------------------------------------------------
---+
only showing top 10 rows
```

Figure 19: Previewing Loaded Text Data

```
>>> language_df = books_df.filter(
   F.col("value"...     F.col("value").startswith("Language:")
... ).withColumn(
gexp_extract("va...     "language",
:\s*(.*)", 1)
)
...     F.regexp_extract("value", r"Language:\s*(.*)", 1)
... )
>>>
>>> language_df.show(5)
[Stage 61:>                                                         (0 + 1)

+----------------+---------+--------+
|           value|file_name|language|
+----------------+---------+--------+
|Language: English|  200.txt| English|
|Language: English|  129.txt| English|
|Language: English|   30.txt| English|
|Language: English|   10.txt| English|
|Language: English|  180.txt| English|
+----------------+---------+--------+
only showing top 5 rows
```

Figure 20: Metadata Extraction Using Regular Expressions(language)

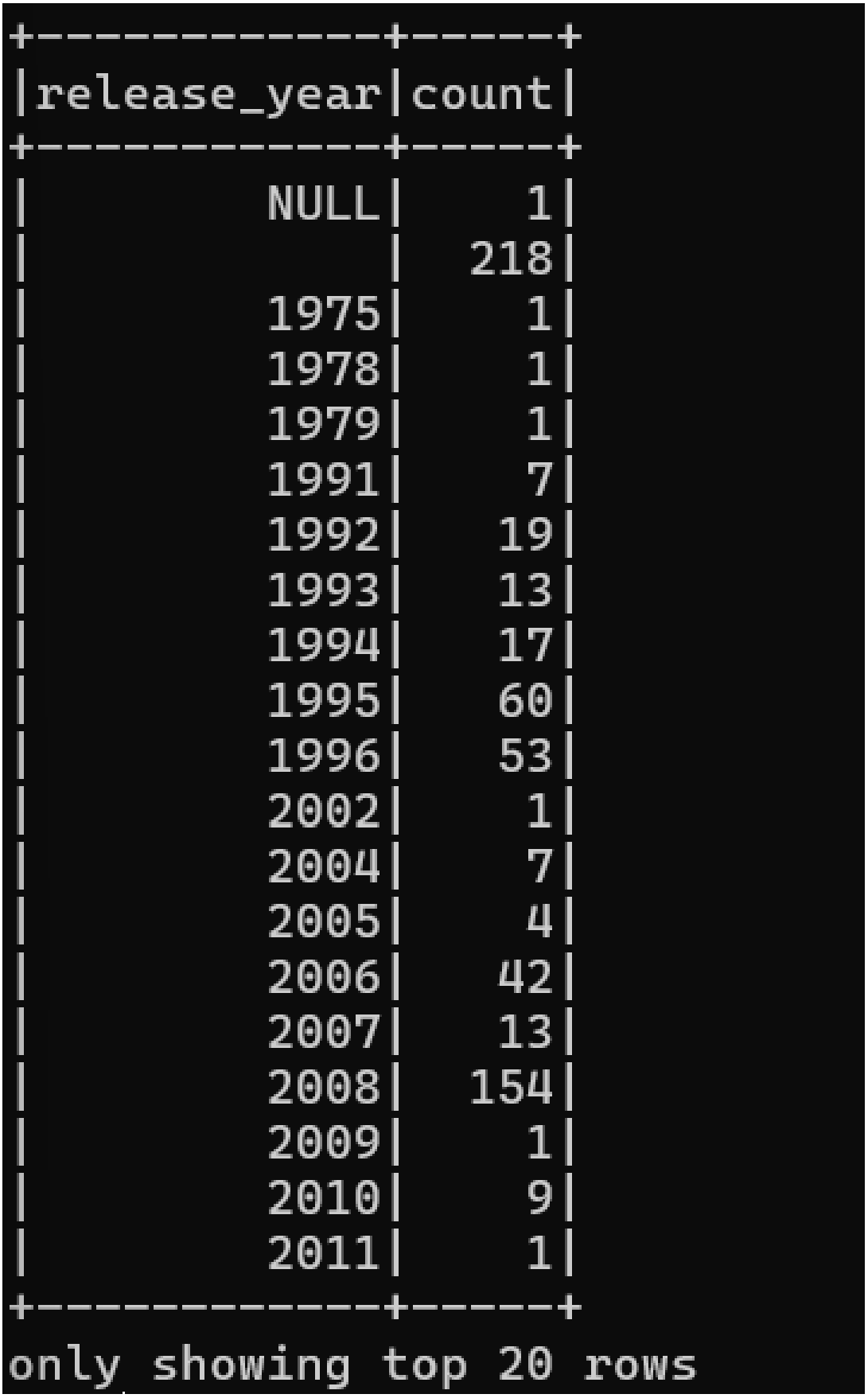
```
>>> release_df = books_df.filter(
  F.col("value")...     F.col("value").startswith("Release Date:")
... ).withColumn(
...     "release_year",
...     F.regexp_extract("value", r"(\d{4})", 1)
... )
>>>
>>> release_df.show(5)
[Stage 60:>                                                         (0 + 1)

+--------------------+---------+------------+
|               value|file_name|release_year|
+--------------------+---------+------------+
|Release Date: Jan...|  200.txt|        1995|
|Release Date: May...|  129.txt|        2008|
|Release Date: Apr...|   30.txt|        1992|
|Release Date: Mar...|   10.txt|        2011|
|Release Date: Jun...|  180.txt|        2008|
+--------------------+---------+------------+
only showing top 5 rows
```

Figure 21: Metadata Extraction Using Regular Expressions(release year)

```
title_df.show(5, truncate=False)
...     F.col("value").startswith("Title:")
... ).withColumn(
...     "title",
...     F.regexp_extract("value", r"Title:\s*(.*)", 1)
... )
>>>
>>> title_df.show(5, truncate=False)
[Stage 59:============================================================(1 + 0)
+-----------------------------------------------------+---------+-----------------------------------------------------+
|value                                                |file_name|title                                                |
+-----------------------------------------------------+---------+-----------------------------------------------------+
|Title: The Project Gutenberg Gutenberg Encyclopedia, Vol 1|200.txt  |The Project Gutenberg Gutenberg Encyclopedia, Vol 1|
|Title: The Square Root of Two, to 5 Million Digits   |129.txt  |The Square Root of Two, to 5 Million Digits          |
|Title: The Bible, King James Version, Complete       |30.txt   |The Bible, King James Version, Complete              |
|Title: The King James Bible                          |10.txt   |The King James Bible                                 |
|Title: The 1994 CIA World Factbook                   |180.txt  |The 1994 CIA World Factbook                          |
+-----------------------------------------------------+---------+-----------------------------------------------------+
only showing top 5 rows
```

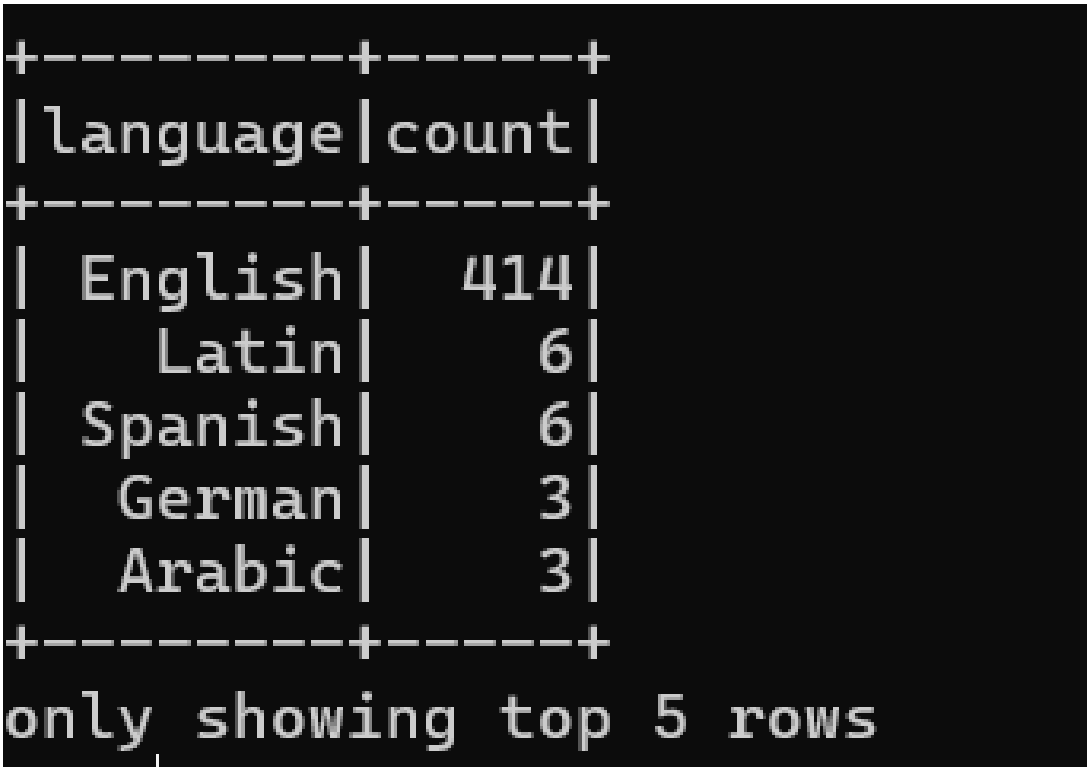
Figure 22: Metadata Extraction Using Regular Expressions(title)



release_year	count
NULL	1
	218
1975	1
1978	1
1979	1
1991	7
1992	19
1993	13
1994	17
1995	60
1996	53
2002	1
2004	7
2005	4
2006	42
2007	13
2008	154
2009	1
2010	9
2011	1

only showing top 20 rows

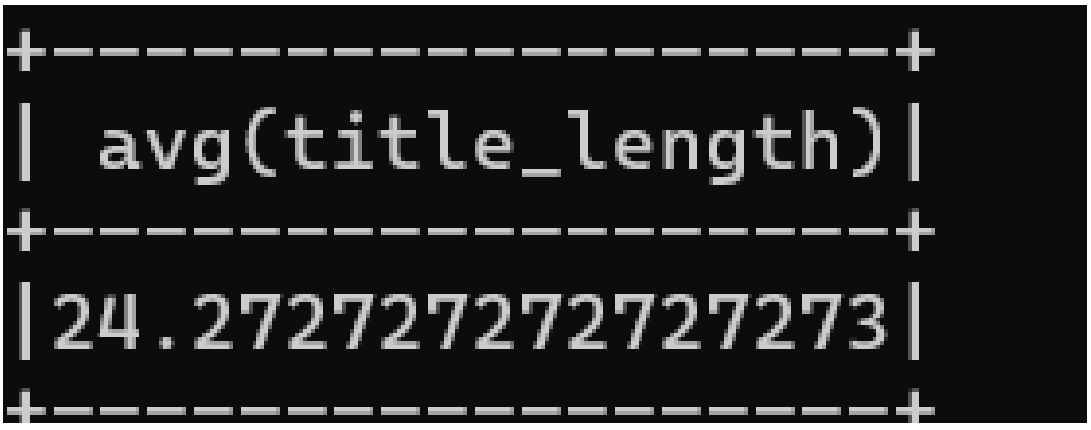
Figure 23: number of books released each year



language	count
English	414
Latin	6
Spanish	6
German	3
Arabic	3

only showing top 5 rows

Figure 24: Most common language



avg(title_length)
24.272727272727273

Figure 25: Average title length

2.2 Question 11: TF–IDF and Cosine Similarity

TF–IDF vectors were computed for each book, and cosine similarity was used to identify similar documents.

Snapshots

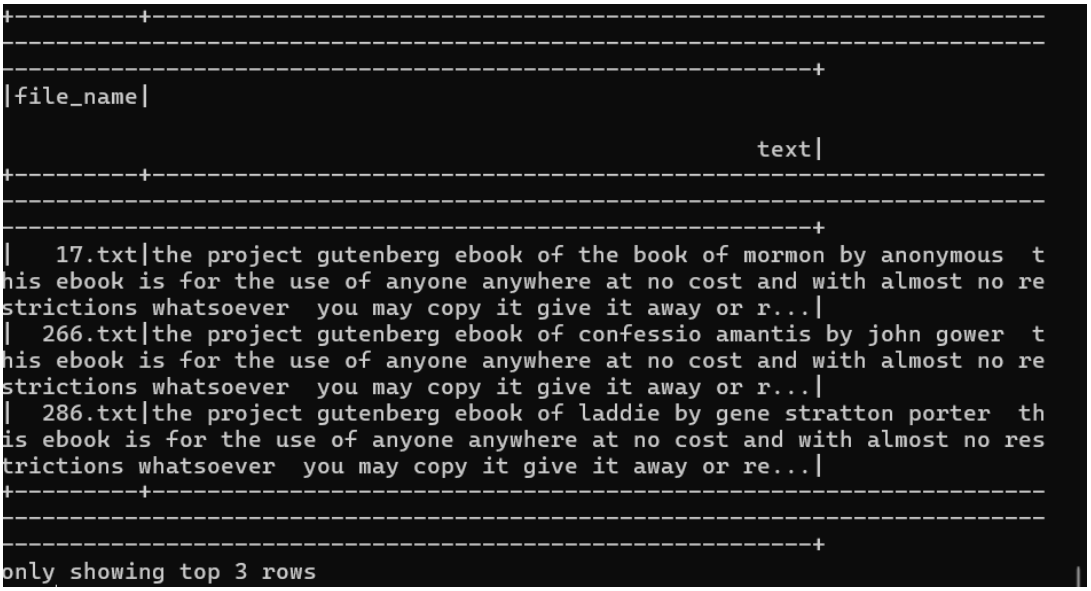


Figure 26: Text cleaning

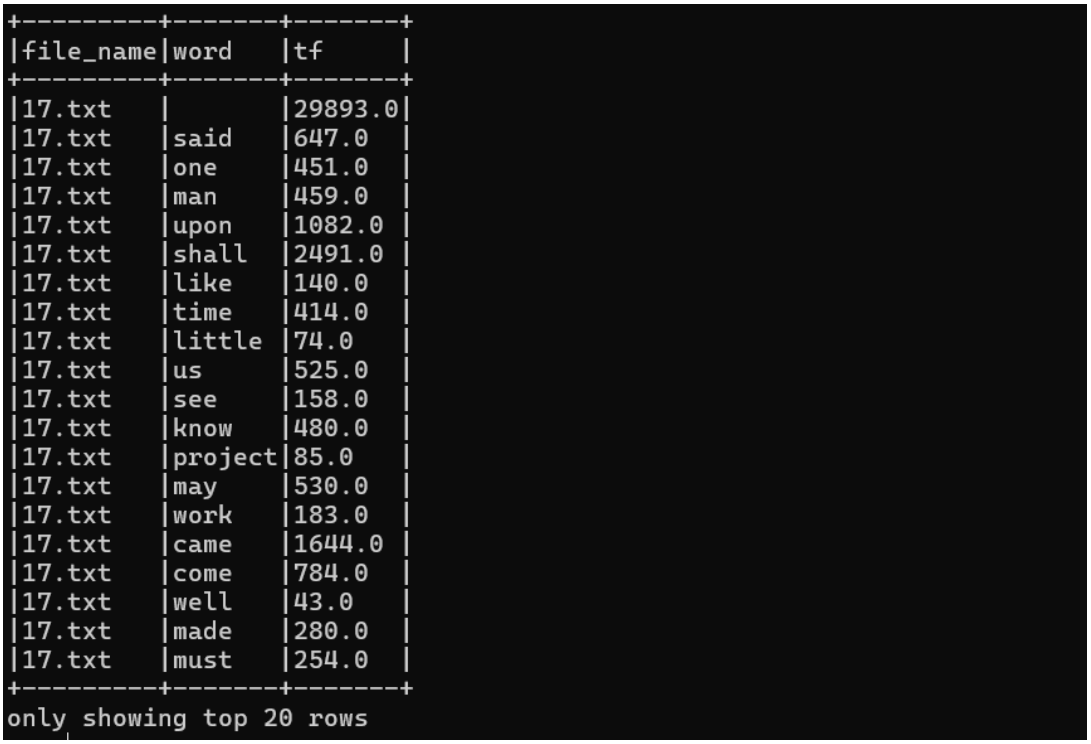
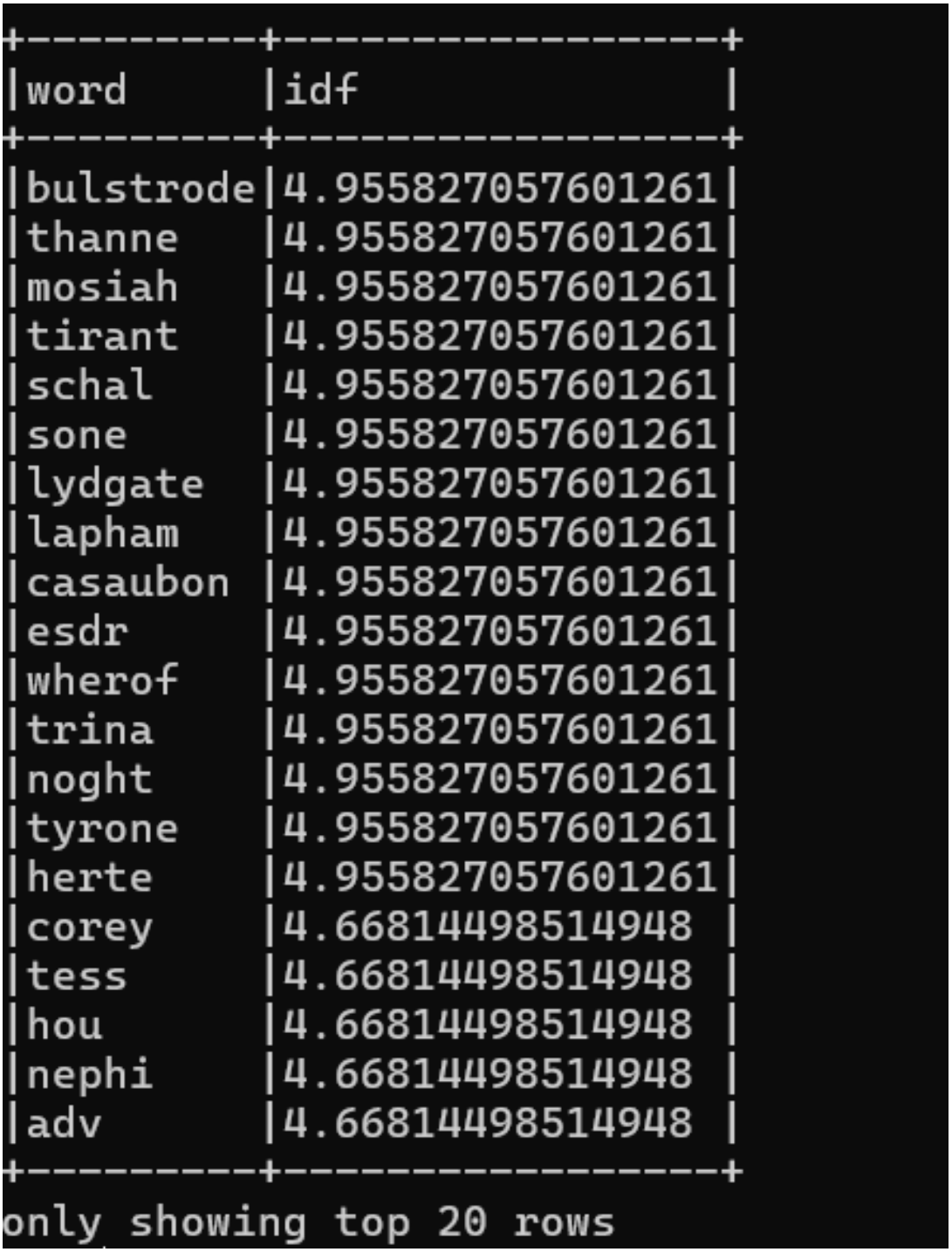


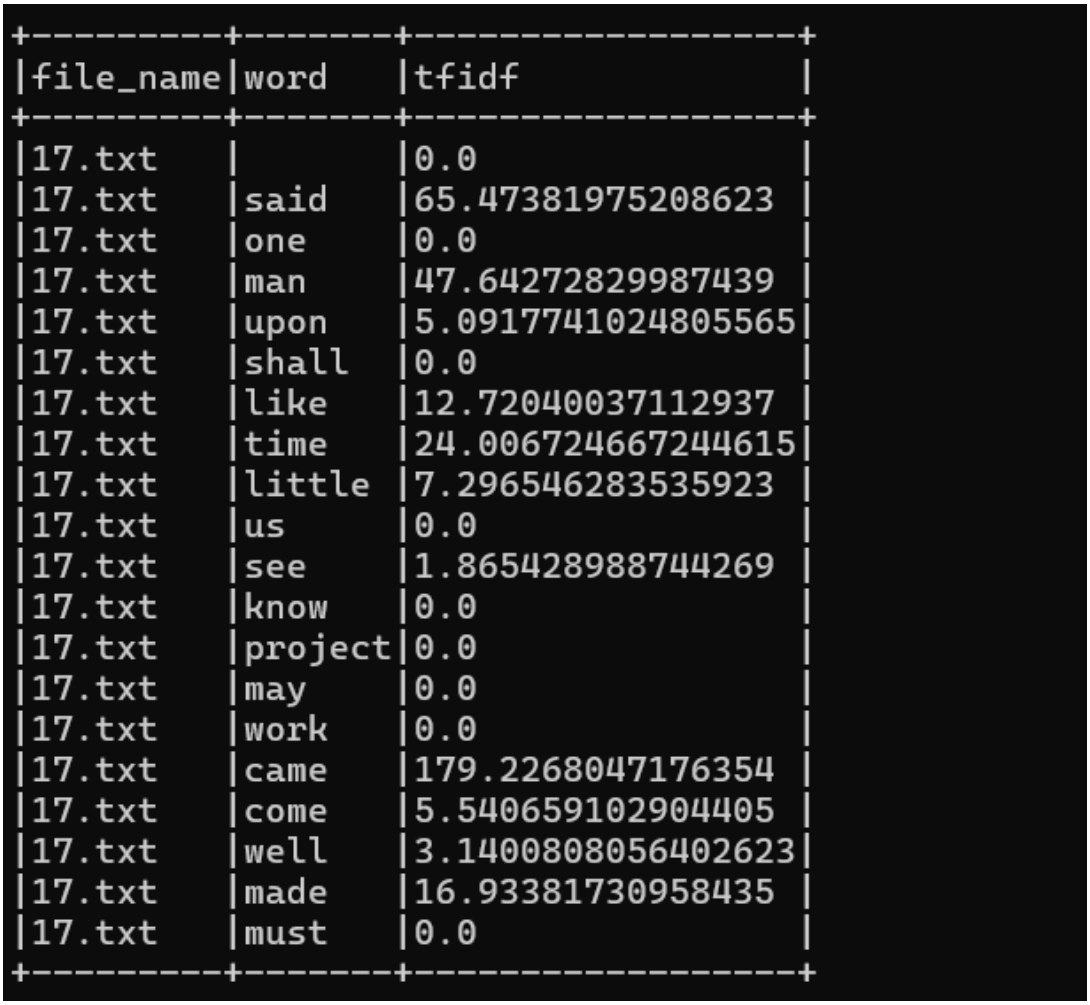
Figure 27: Term Frequency (TF)



word	idf
bulstrode	4.955827057601261
thanne	4.955827057601261
mosiah	4.955827057601261
tirant	4.955827057601261
schal	4.955827057601261
sone	4.955827057601261
lydgate	4.955827057601261
lapham	4.955827057601261
casaubon	4.955827057601261
esdr	4.955827057601261
wherof	4.955827057601261
trina	4.955827057601261
noght	4.955827057601261
tyrone	4.955827057601261
herte	4.955827057601261
corey	4.668144985149 48
tess	4.668144985149 48
hou	4.668144985149 48
nephi	4.668144985149 48
adv	4.668144985149 48

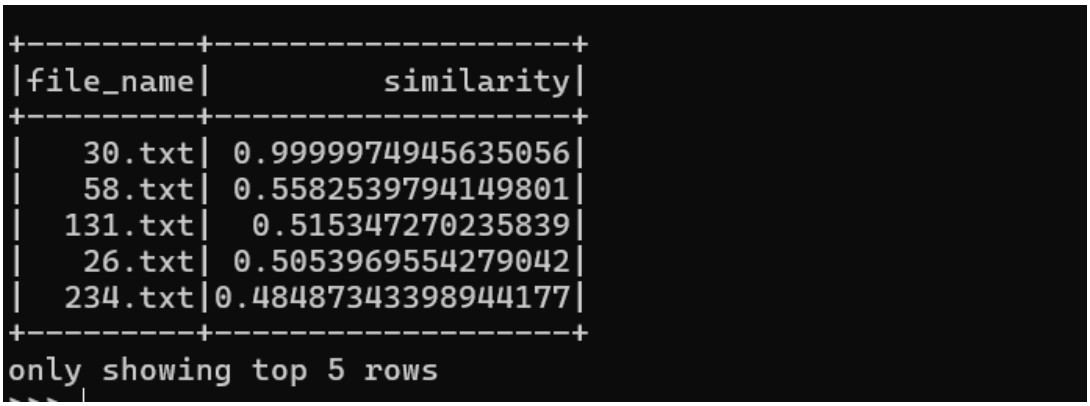
only showing top 20 rows

Figure 28: Inverse Document Frequency (IDF)



file_name	word	tfidf
17.txt		0.0
17.txt	said	65.47381975208623
17.txt	one	0.0
17.txt	man	47.64272829987439
17.txt	upon	5.0917741024805565
17.txt	shall	0.0
17.txt	like	12.72040037112937
17.txt	time	24.006724667244615
17.txt	little	7.296546283535923
17.txt	us	0.0
17.txt	see	1.865428988744269
17.txt	know	0.0
17.txt	project	0.0
17.txt	may	0.0
17.txt	work	0.0
17.txt	came	179.2268047176354
17.txt	come	5.540659102904405
17.txt	well	3.1400808056402623
17.txt	made	16.93381730958435
17.txt	must	0.0

Figure 29: TF–IDF computation



file_name	similarity
30.txt	0.9999974945635056
58.txt	0.5582539794149801
131.txt	0.515347270235839
26.txt	0.5053969554279042
234.txt	0.48487343398944177

only showing top 5 rows

Figure 30: Cosine Similarity Between Documents

2.3 Question 12: Author Influence Network

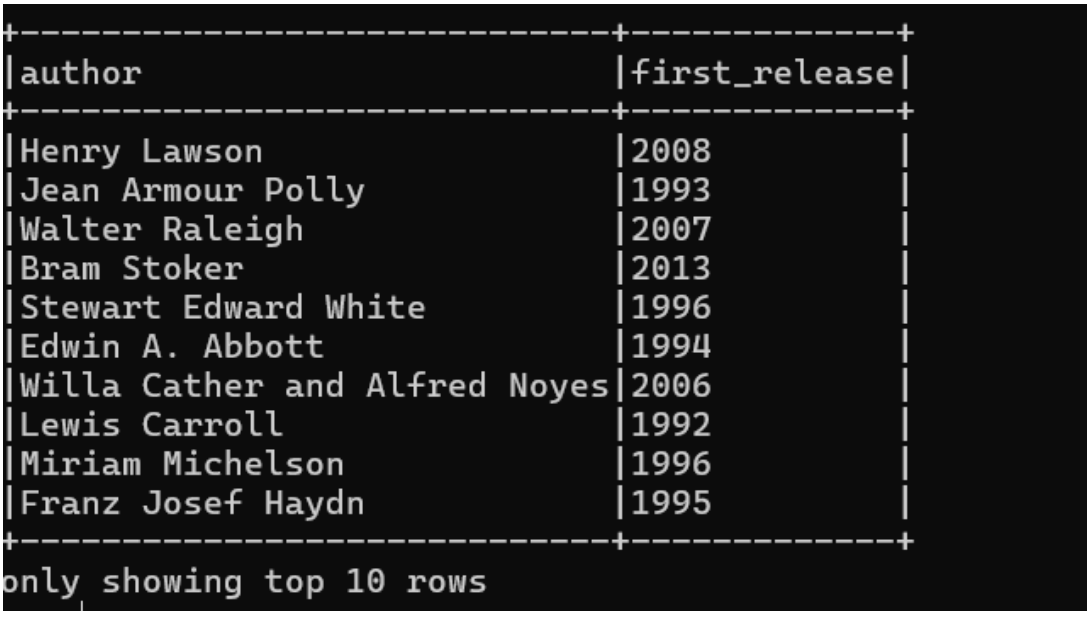
To construct the author influence network, author–year metadata was duplicated with renamed columns to avoid ambiguity during self-joins. A directed edge was created from a source author to a target author when the source author's publication year was earlier than the target author's year. This temporal ordering models potential literary influence between authors.

Snapshots

file_name	author	release_year
17.txt	Anonymous	1992
266.txt	John Gower	2008
22.txt	Peter Mark Roget	1991
129.txt	Robert Nemiroff	2008
30.txt	Various	1992
48.txt	United States. C...	1993
200.txt	Project Gutenberg	1995
145.txt	George Eliot	2008
450.txt	David Graham Phil...	2006
25.txt	United States. C...	1992

only showing top 10 rows

Figure 31: Extracted author and year metadata



author	first_release
Henry Lawson	2008
Jean Armour Polly	1993
Walter Raleigh	2007
Bram Stoker	2013
Stewart Edward White	1996
Edwin A. Abbott	1994
Willa Cather and Alfred Noyes	2006
Lewis Carroll	1992
Miriam Michelson	1996
Franz Josef Haydn	1995

only showing top 10 rows

Figure 32: Extracted author and first release

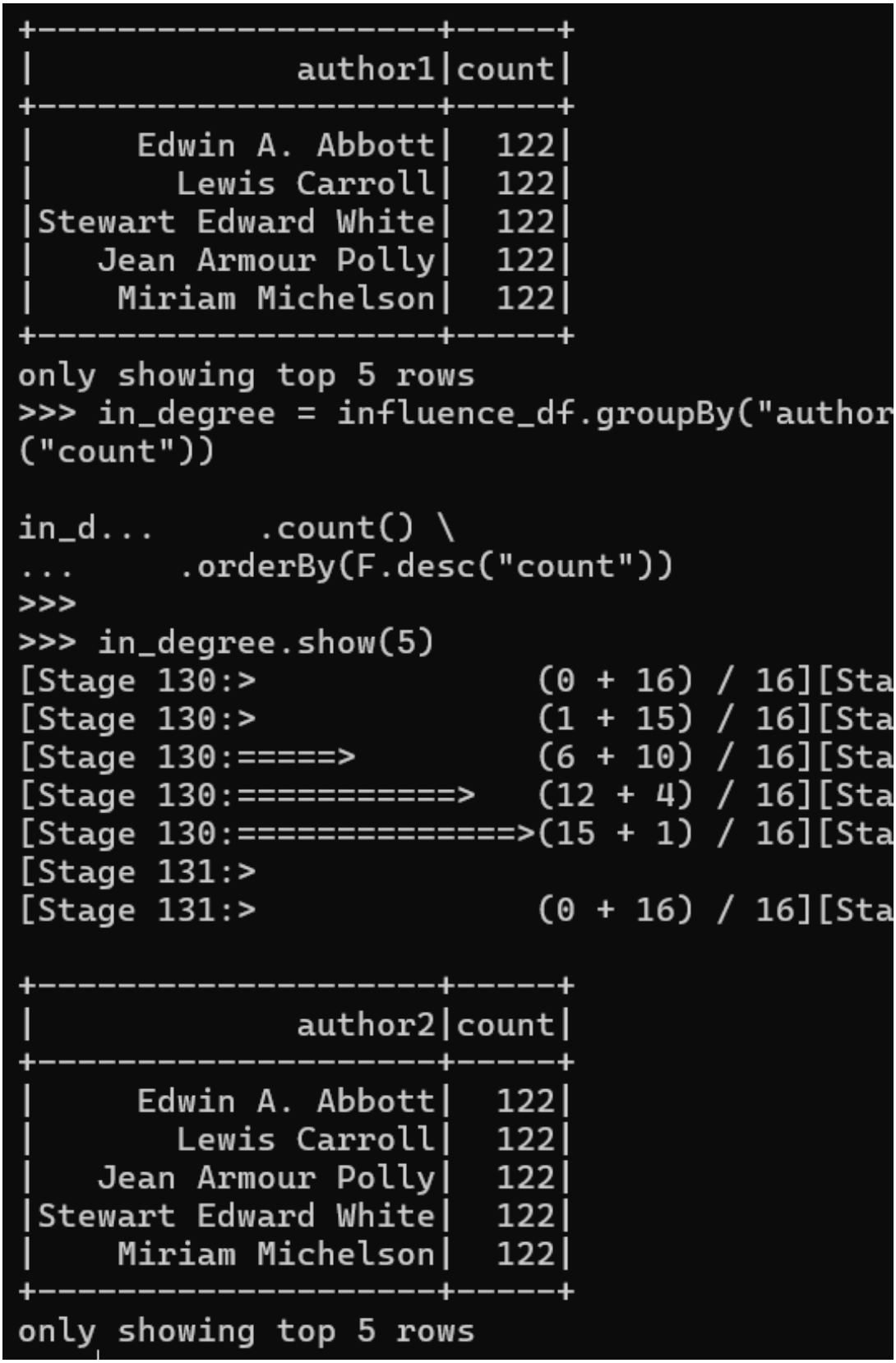


Figure 33: Author influence analysis

3 Conclusion

This assignment provided hands-on experience with Hadoop MapReduce and Apache Spark for large-scale text analytics. The experiments demonstrated distributed storage, parallel computation, and advanced analytics.