Assignment 1: MapReduce and Apache Spark

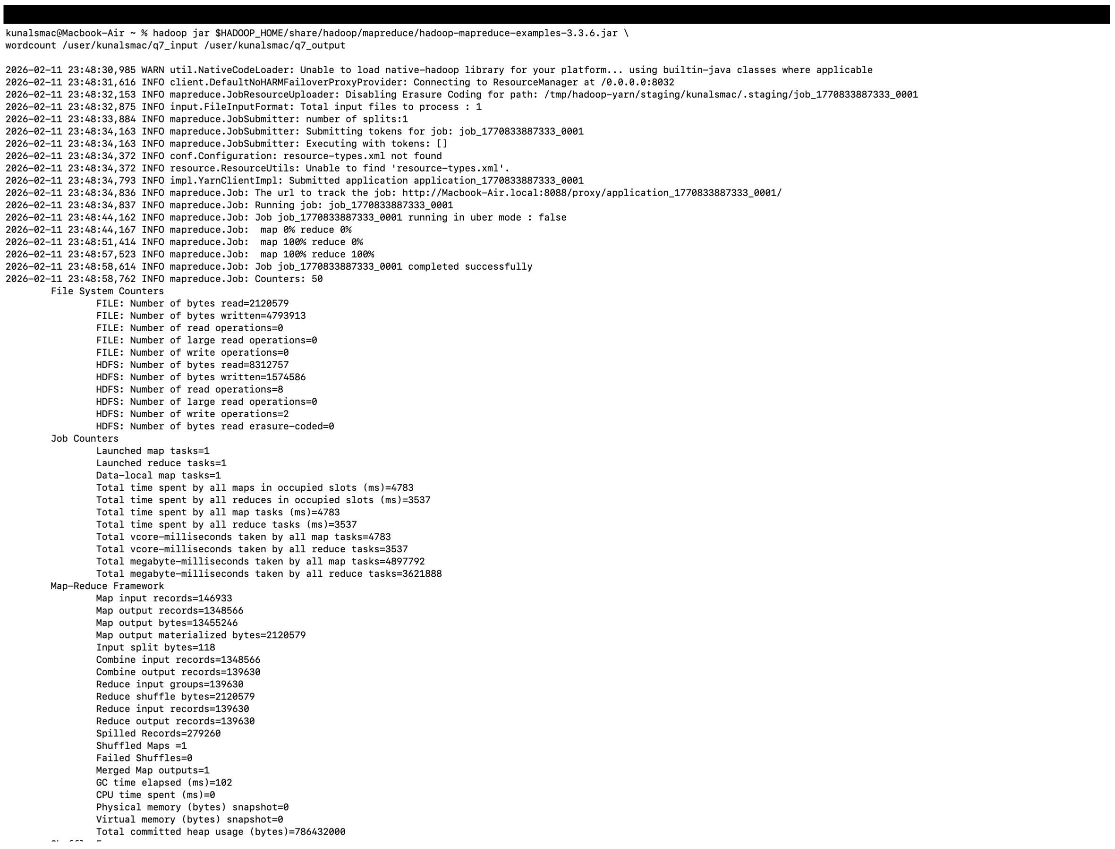
Name: Kunal Kumar Mishra
Roll No: M25CSA036
Program: M.Tech AI
Course: Machine Learning with Big Data (CSL7110)
Github Repository : Repository Link

Question 1:

Execute the Apache Hadoop WordCount example and present the output to demonstrate its working.

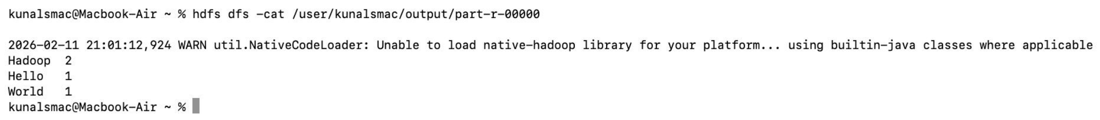
Answer:

The WordCount example provided by Apache Hadoop was executed on the local system to demonstrate the working of the MapReduce framework. The program was run using the Hadoop MapReduce examples JAR file. The MapReduce job executed successfully, and the output containing the frequency of each word was generated in HDFS. This confirms that the WordCount example is working correctly as expected.



```
kunalsmac@Macbook-Air ~ % hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.6.jar \
wordcount /user/kunalsmac/q7_input /user/kunalsmac/q7_output

2026-02-11 23:48:30,985 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2026-02-11 23:48:31,616 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2026-02-11 23:48:32,153 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/kunalsmac/.staging/job_1770833887333_0001
2026-02-11 23:48:32,875 INFO input.FileInputFormat: Total input files to process : 1
2026-02-11 23:48:33,884 INFO mapreduce.JobSubmitter: number of splits:1
2026-02-11 23:48:34,163 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1770833887333_0001
2026-02-11 23:48:34,163 INFO mapreduce.JobSubmitter: Executing with tokens: []
2026-02-11 23:48:34,372 INFO conf.Configuration: resource-types.xml not found
2026-02-11 23:48:34,372 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2026-02-11 23:48:34,793 INFO impl.YarnClientImpl: Submitted application application_1770833887333_0001
2026-02-11 23:48:34,836 INFO mapreduce.Job: The url to track the job: http://Macbook-Air.local:8088/proxy/application_1770833887333_0001/
2026-02-11 23:48:34,837 INFO mapreduce.Job: Running job: job_1770833887333_0001
2026-02-11 23:48:44,162 INFO mapreduce.Job: Job job_1770833887333_0001 running in uber mode : false
2026-02-11 23:48:44,167 INFO mapreduce.Job:  map 0% reduce 0%
2026-02-11 23:48:51,414 INFO mapreduce.Job:  map 100% reduce 0%
2026-02-11 23:48:57,523 INFO mapreduce.Job:  map 100% reduce 100%
2026-02-11 23:48:58,614 INFO mapreduce.Job: Job job_1770833887333_0001 completed successfully
2026-02-11 23:48:58,762 INFO mapreduce.Job: Counters: 50
	File System Counters
		FILE: Number of bytes read=2120579
		FILE: Number of bytes written=4793913
		FILE: Number of read operations=0
		FILE: Number of large read operations=0
		FILE: Number of write operations=0
		HDFS: Number of bytes read=8312757
		HDFS: Number of bytes written=1574586
		HDFS: Number of read operations=8
		HDFS: Number of large read operations=0
		HDFS: Number of write operations=2
		HDFS: Number of bytes read erasure-coded=0
	Job Counters
		Launched map tasks=1
		Launched reduce tasks=1
		Data-local map tasks=1
		Total time spent by all maps in occupied slots (ms)=4783
		Total time spent by all reduces in occupied slots (ms)=3537
		Total time spent by all map tasks (ms)=4783
		Total time spent by all reduce tasks (ms)=3537
		Total vcore-milliseconds taken by all map tasks=4783
		Total vcore-milliseconds taken by all reduce tasks=3537
		Total megabyte-milliseconds taken by all map tasks=4897792
		Total megabyte-milliseconds taken by all reduce tasks=3621888
	Map-Reduce Framework
		Map input records=146933
		Map output records=1348566
		Map output bytes=13455246
		Map output materialized bytes=2120579
		Input split bytes=118
		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)=102
		CPU time spent (ms)=0
		Physical memory (bytes) snapshot=0
		Virtual memory (bytes) snapshot=0
		Total committed heap usage (bytes)=786432000
```

Figure 1: Successful execution of the WordCount MapReduce Job



```
kunalsmac@Macbook-Air ~ % hdfs dfs -cat /user/kunalsmac/output/part-r-00000

2026-02-11 21:01:12,924 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Hadoop  2
Hello   1
World   1
kunalsmac@Macbook-Air ~ %
```

Figure 2: Output generated by the WordCount Program in HDFS

Question 2:

Analyze the Map phase of the Hadoop WordCount example for the given song lyrics, including the mapper output pairs and the data types of input and output keys and values used in the Map phase (Hadoop I/O types).

Answer:

In the Map phase of the WordCount program, each line of the input file is processed independently by the Mapper. The input key represents the byte offset of the line from the beginning of the file, and the input value represents the line of text.

The Mapper splits each line into individual words and emits an intermediate key–value pair for each word. Each emitted pair consists of the word as the key and the value 1, indicating one occurrence of that word.

Hence, the output pairs generated by the Mapper will be of the form (word, 1), for example: (up, 1), (all, 1), (night, 1), (sun, 1), (get, 1), (lucky, 1), and so on for all words in the input.

The data types of the input and output pairs in the Map phase are as follows:

- Input Key: LongWritable (reprsents the byte offset of the line)
- Input Value: Text (represnets a line of text)
- Output Key: Text (represents a word)
- Output Value: IntWritable (reprsents the count of the word)

These data types belong to the org.apache.hadoop.io package and are used by Hadoop instead of standard Java data types.

Question 3:

Reduce phase input–output pairs and data types in WordCount

Answer:

In the Reduce phase of the WordCount program, the Reducer receives intermediate data produced by the Mapper after the shuffle and sort phase. During this process, all values corresponding to the same key are grouped together and sent to a single Reducer.

For example, if the Mapper emits multiple occurrences of the word “up” as (up, 1), the Reducer will receive the input pair in the form (up, [1, 1, 1, 1]). Similarly, other words such as “to”, “get”, and “lucky” will also be grouped with their respective lists of values.

Thus, the input pairs to the Reduce phase consist of a word as the key and a list of integer values representing the number of times the word appeared in the Map phase.

The Reducer processes each input pair by summing all the values associated with a given key and emits a final output pair containing the word and its total count. For example, the output pairs generated by the Reduce phase include (up, 4), (to, 3), (get, 2), and (lucky, 1).

The data types used in the Reduce phase are as follows:

- Input Key: Text
- Input Value: Iterable<IntWritable>
- Output Key: Text
- Output Value: IntWritable

These data types belong to the org.apache.hadoop.io package and ensure efficient serialization and processing within the Hadoop framework.

Question 4:

Data types used in Mapper and Reducer of WordCount.

Answer:

In the WordCount program provided in the Apache Hadoop MapReduce Tutorial, the Mapper and Reducer classes use Hadoop-specific data types from the org.apache.hadoop.io package instead of standard Java data types.

1. Mapper Class and map( ) Function

From the WordCount source code:

```
public static class TokenizerMapper

    extends Mapper<Object, Text, Text, IntWritable> {



  public void map(Object key, Text value, Context context)

       throws IOException, InterruptedException {

    ..}.

  }
```

Replaced data types in the map ( ) function are:

- Input Key Type: Object
- Input Value Type: Text
- Output Key type: Text
- Output Value type: IntWritable

Explanation:

1) The input key (Object) represents the **byte offset** of the line in the input file.
2) The input value (Text) represents one line of text read from the input file.
3) The mapper emits each **word** as a key (Text) with a count of **1** as the value (IntWritable).
4) Text and IntWritable are Hadoop's serializable data types that implement the Writable interface.

2. Reducer Class and reduce ( ) Function

From the WordCount source code:

```
public static class IntSumReducer

    extends Reducer<Text, IntWritable, Text, IntWritable> {



  public void reduce(Text key, Iterable<IntWritable> values,

                   Context context)

        throws IOException, InterruptedException {

    ...

  }

}
```

Replaced data types in the reduce ( ) function are:

- Input Key type: Text
- Input Value type: Iterable<IntWritable>
- Output Key Type: Text
- Output value type: IntWritable

Explanation:

The reducer receives a **word** (Text) as the key.

The values (Iterable<IntWritable>) are all the counts associated with that word from different mappers.
The reducer sums these values and outputs:

i. the word (Text)
ii. its total count (IntWritable)

Conclusion:

The WordCount example uses Hadoop's Writable data types to ensure efficient serialization and communication between Mapper and Reducer tasks. The Mapper transforms lines of text into intermediate (word, 1) pairs, while the Reducer aggregates these values to compute the final word frequencies.

Question 5 :

Explanation of the Mapper logic used in WordCount.

Answer:

The Mapper phase in the WordCount program is responsible for processing the input data and generating intermediate key–value pairs.

Each input to the Mapper is a pair consisting of a key and a value:

- The **key** represents the byte offset of the line in the input file
- The value represents a single line of text from the input file

The Mapper performs the following operations:

1. The input line is first converted to lowercase to avoid case-sensitive duplicates (for example, treating “Hadoop” and “hadoop” as different words).
2. Punctuation and unwanted characters are removed using a regular expression.
3. The cleaned line is split into individual words using whitespace as the delimiter.
4. For each word, the Mapper emits an intermediate key–value pair of the form **(word, 1)**.

Here, the word is represented using Hadoop’s **Text** data type, and the count value **1** is represented using **IntWritable**.

This Mapper output prepares the data for the Reduce phase, where all values associated with the same word are grouped together and summed to compute the total frequency of each word.

Question 6

Explanation of the Reduce logic used in WordCount.

Answer:

The Reduce phase in the WordCount program is responsible for aggregating the intermediate results produced by the Mapper phase.

Each input to the Reducer is a key-value pair of the form:

- Key: a word (Text)
- value: an iterable list of counts (Iterable<IntWritable>) associated with that word

All values corresponding to the same key (word) are grouped together by the Hadoop framework before being passed to the Reducer.

The Reducer performs the following operations:

1. For a given word (key), it receives a list of integer values, where each value represents an occurrence of that word emitted by the Mapper.
2. The Reducer iterates through the list of values and computes their sum.
3. The final output is emitted as a key-value pair (word, total_count).

Here, the word is represented using the Text data type and the final count is represented using IntWritable, both from the org.apache.hadoop.io package.

The Reduce phase produces the final result of the WordCount job, which contains each unique word along with the total number of times it appears in the input dataset.

Question 7:

WordCount on dataset file 200.txt

Answer:

For this question, the dataset containing multiple text files was first downloaded and extracted on the local system. As instructed, the file **200.txt** was selected from the dataset and used as input for the Hadoop MapReduce job.

Step 1: Creating an Input Directory in HDFS

A new directory was created in HDFS to store the input file for Question 7. This directory is used specifically for processing the file *200.txt*.

Step 2: Copying the File to HDFS

The file **200.txt** was copied from the local file system (Desktop) into the HDFS input directory using the hdfs dfs -put command. This step ensures that the input data is available in HDFS for distributed processing by Hadoop.

Step 3: Running the WordCount MapReduce Job

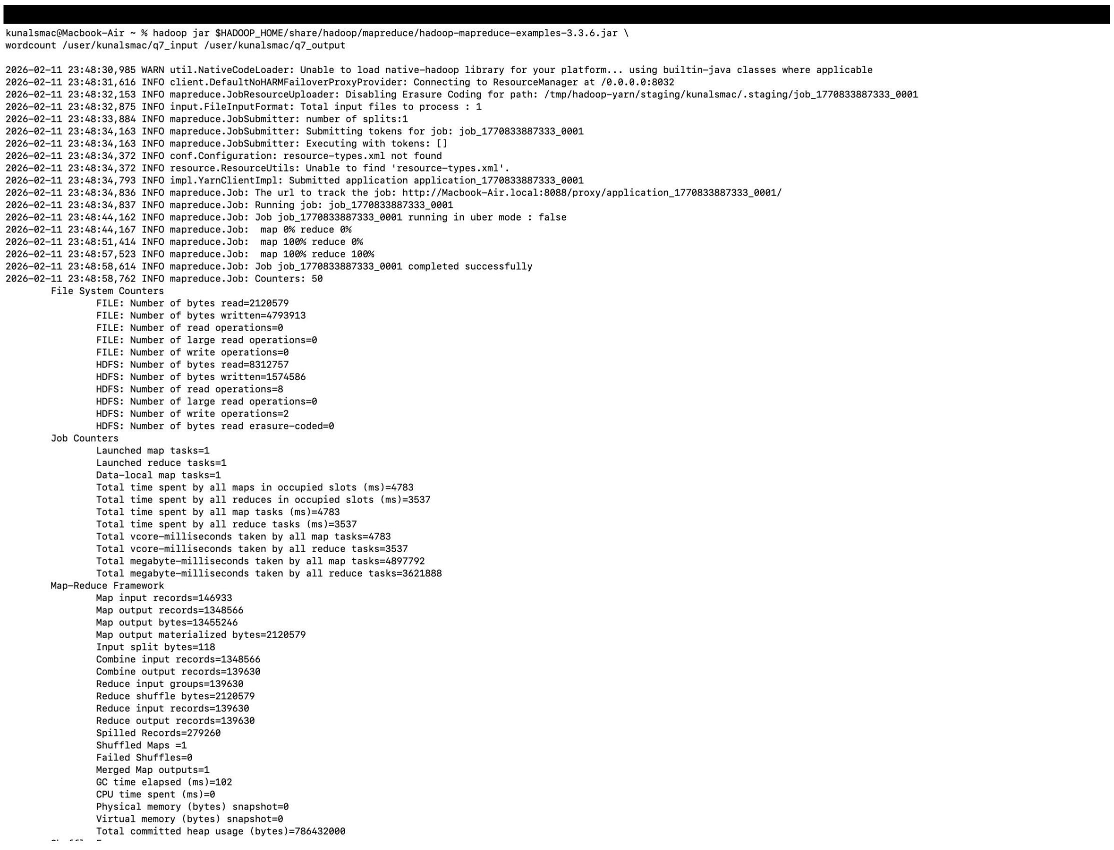
The WordCount example program provided with Hadoop was executed on the input file **200.txt**. The job processes the file by splitting the text into words during the Map phase and counting the occurrences of each word during the Reduce phase.
The output of the job was written to a separate output directory in HDFS.

Step 4: Verifying the Output

After successful execution of the MapReduce job, the output file (part-r-00000) was inspected. The output contains word–count pairs, where each word from the input file appears along with the number of times it occurs in the document.

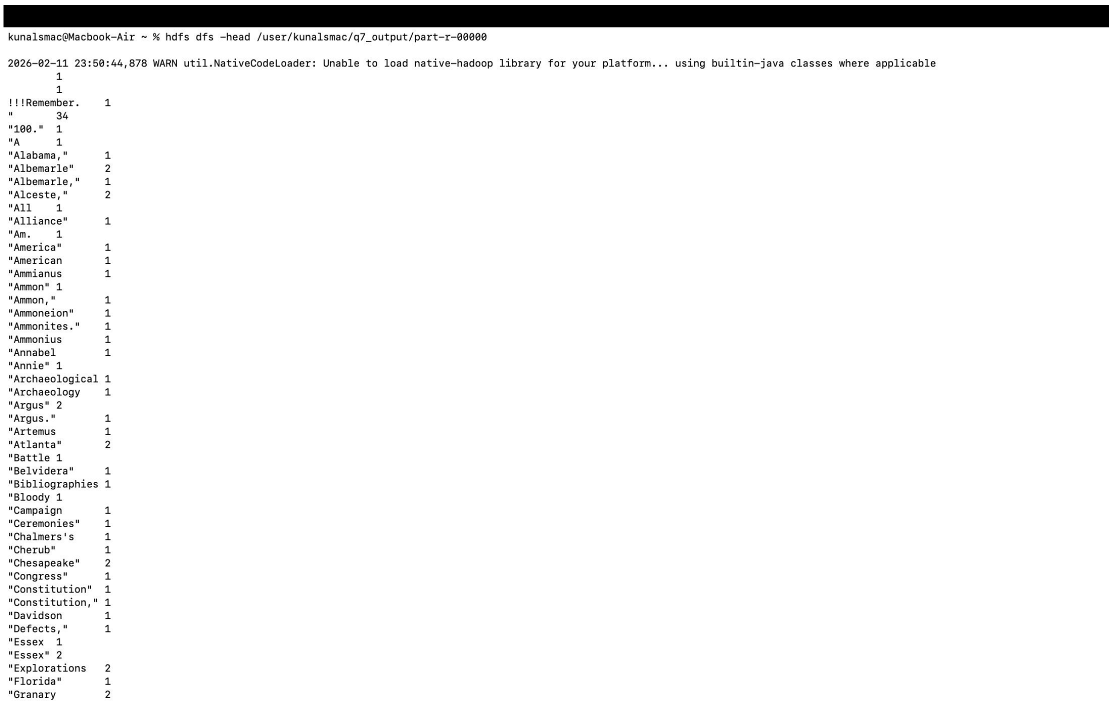
Result

The successful execution of the WordCount job on **200.txt** confirms that Hadoop MapReduce is correctly configured and functioning as expected. The output demonstrates correct tokenization of text and accurate aggregation of word frequencies.



```
kunalsmac@Macbook-Air ~ % hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.6.jar \
wordcount /user/kunalsmac/q7_input /user/kunalsmac/q7_output

2026-02-11 23:48:30,985 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2026-02-11 23:48:31,616 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2026-02-11 23:48:32,153 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/kunalsmac/.staging/job_1770833887333_0001
2026-02-11 23:48:32,875 INFO input.FileInputFormat: Total input files to process : 1
2026-02-11 23:48:33,884 INFO mapreduce.JobSubmitter: number of splits:1
2026-02-11 23:48:34,163 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1770833887333_0001
2026-02-11 23:48:34,163 INFO mapreduce.JobSubmitter: Executing with tokens: []
2026-02-11 23:48:34,372 INFO conf.Configuration: resource-types.xml not found
2026-02-11 23:48:34,372 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2026-02-11 23:48:34,793 INFO impl.YarnClientImpl: Submitted application application_1770833887333_0001
2026-02-11 23:48:34,836 INFO mapreduce.Job: The url to track the job: http://Macbook-Air.local:8088/proxy/application_1770833887333_0001/
2026-02-11 23:48:34,837 INFO mapreduce.Job: Running job: job_1770833887333_0001
2026-02-11 23:48:44,162 INFO mapreduce.Job: Job job_1770833887333_0001 running in uber mode : false
2026-02-11 23:48:44,167 INFO mapreduce.Job:  map 0% reduce 0%
2026-02-11 23:48:51,414 INFO mapreduce.Job:  map 100% reduce 0%
2026-02-11 23:48:57,523 INFO mapreduce.Job:  map 100% reduce 100%
2026-02-11 23:48:58,614 INFO mapreduce.Job: Job job_1770833887333_0001 completed successfully
2026-02-11 23:48:58,762 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=2120579
                FILE: Number of bytes written=4793913
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=8312757
                HDFS: Number of bytes written=1574586
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=4783
                Total time spent by all reduces in occupied slots (ms)=3537
                Total time spent by all map tasks (ms)=4783
                Total time spent by all reduce tasks (ms)=3537
                Total vcore-milliseconds taken by all map tasks=4783
                Total vcore-milliseconds taken by all reduce tasks=3537
                Total megabyte-milliseconds taken by all map tasks=4897792
                Total megabyte-milliseconds taken by all reduce tasks=3621888
        Map-Reduce Framework
                Map input records=146933
                Map output records=1348566
                Map output bytes=13455246
                Map output materialized bytes=2120579
                Input split bytes=118
                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)=102
                CPU time spent (ms)=0
                Physical memory (bytes) snapshot=0
                Virtual memory (bytes) snapshot=0
                Total committed heap usage (bytes)=786432000
```

Fig 7.1: Execution of wordcount MapReduce job on 200.txt



```
kunalsmac@Macbook-Air ~ % hdfs dfs -head /user/kunalsmac/q7_output/part-r-00000

2026-02-11 23:50:44,878 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
        1
        1
!!!Remember.    1
"       34
"100."  1
"A      1
"Alabama,"      1
"Albemarle"     2
"Albemarle,"    1
"Alceste,"      2
"All    1
"Alliance"      1
"Am.    1
"America"       1
"American       1
"Ammianus       1
"Ammon" 1
"Ammon,"        1
"Ammoneion"     1
"Ammonites."    1
"Ammonius       1
"Annabel        1
"Annie" 1
"Archaeological 1
"Archaeology    1
"Argus" 2
"Argus."        1
"Artemus        1
"Atlanta"       2
"Battle 1
"Belvidera"     1
"Bibliographies 1
"Bloody 1
"Campaign       1
"Ceremonies"    1
"Chalmers's     1
"Cherub"        1
"Chesapeake"    2
"Congress"      1
"Constitution"  1
"Constitution," 1
"Davidson       1
"Defects,"      1
"Essex  1
"Essex" 2
"Explorations   2
"Florida"       1
"Granary        2
```

Fig 7.2 : Sample output of WordCont job showing word frequencies

Question 8: HDFS File System Commands and Replication

Answer:

HDFS provides a file system interface similar to UNIX, where common file system operations such as listing directories, copying files, and removing files can be performed using the hadoop fs command. For example, files from the local file system can be copied to HDFS using the -copyFromLocal option, and directory contents can be viewed using the -ls option.

The output of the hadoop fs -ls command is similar to the UNIX ls command. However, an additional column is present that shows the **replication factor** of files stored in HDFS. The replication factor indicates how many copies of a file's data blocks are stored across different DataNodes in the cluster to provide fault tolerance and reliability.

Directories in HDFS do **not** have a replication factor because they do not store actual data blocks. A directory in HDFS only contains **metadata**, such as file names, permissions, and references to file blocks. This metadata is maintained by the **NameNode**, not by DataNodes.

Replication is applied only to **file data blocks**, which are distributed across multiple DataNodes to ensure data availability in case of node failures. Since directories do not contain data blocks and only serve as a logical structure for organizing files, replication is neither required nor applicable for directories.

Question 9: Measuring Job Execution Time and Input Split Size Impact

Answer:

To measure the total execution time of the WordCount MapReduce job, the system time can be recorded immediately before and after the job execution. This is achieved by capturing the current system time before calling job.waitForCompletion(true) and again after the job finishes. The difference between these two timestamps gives the total execution time of the job in milliseconds.

This approach measures the **end-to-end execution time**, including job initialization, task scheduling, map execution, shuffle and sort, reduce execution, and job completion.

The parameter mapreduce.input.fileinputformat.split.maxsize controls the **maximum size of an input split**. Input splits determine how many mapper tasks are created for a job. By changing this parameter, we can control the number of map tasks spawned during execution.

When the split size is **reduced**, the input file is divided into a larger number of smaller splits, resulting in **more mapper tasks**. This increases parallelism but also introduces overhead due to task creation, scheduling, and context switching. If the splits are too small, the overhead can outweigh the benefits of parallel execution, leading to degraded performance.

When the split size is **increased**, fewer and larger input splits are created, resulting in **fewer mapper tasks**. This reduces task management overhead but may lead to underutilization of cluster resources, as fewer mappers run in parallel.

Therefore, performance is optimal when the split size balances **parallelism and overhead**. The ideal value depends on factors such as input size, cluster resources, and workload characteristics. This explains why adjusting `mapreduce.input.fileinputformat.split.maxsize` directly impacts job execution time and overall performance.

Question 10: Book Metadata Extraction and Analysis Using Apache Spark

Answer:

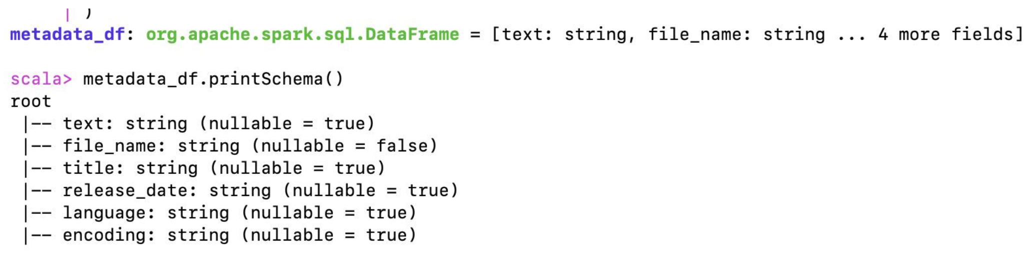
10.1 Metadata Extraction

The Project Gutenberg dataset was processed using Apache Spark. All text files were loaded into a Spark DataFrame named books_df, where each row represents a complete book. The wholetext option was used to ensure that the entire content of each book was read as a single record instead of being split line by line.

From the text column of each book, the following metadata fields were extracted:

- Title
- Release Date
- Language
- Character Set Encoding

Metadata extraction was performed using Spark SQL DataFrame transformations and regular expressions. A new DataFrame named `metadata_df` was created by applying successive column transformations to extract the required metadata fields from the raw text.



```
     | )
metadata_df: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 4 more fields]

scala> metadata_df.printSchema()
root
 |-- text: string (nullable = true)
 |-- file_name: string (nullable = false)
 |-- title: string (nullable = true)
 |-- release_date: string (nullable = true)
 |-- language: string (nullable = true)
 |-- encoding: string (nullable = true)
```

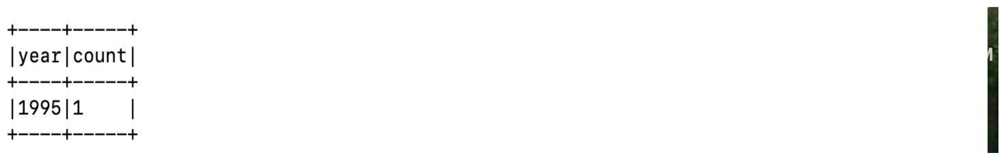
Fig 10.1: Schema of metadata_df showoing extracted book metadata

10.2 Analysis of Extracted Metadata

10.2.1 Number of Books Released Each year

To analyze publication trends, the release year was extracted from the release_date field using a regular expression. The dataset was then grouped by year, and the total number of books released in each year was calculated using Spark's groupBy and count operations.

This analysis provides insights into how book releases are distributed across different years in the dataset.



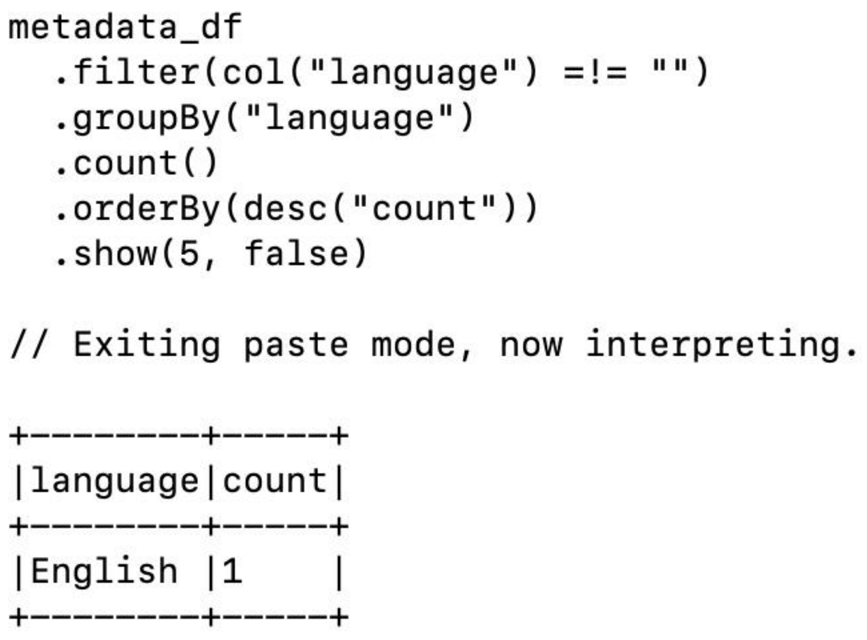
```
+----+-----+
|year|count|
+----+-----+
|1995|1    |
+----+-----+
```

Fig 10.2: Number of Books released per year

10.2.2 Most Common Language in the Dataset

The most common language in the dataset was determined by grouping the extracted language field and counting the number of books available in each language. The results were sorted in descending order to identify the language with the highest frequency.

This analysis highlights the dominant language present in the Project Gutenberg dataset.



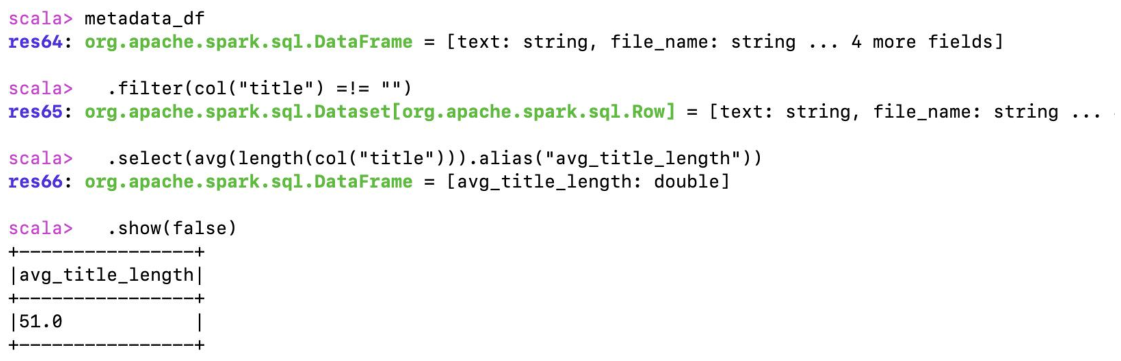
```
metadata_df
  .filter(col("language") =!= "")
  .groupBy("language")
  .count()
  .orderBy(desc("count"))
  .show(5, false)

// Exiting paste mode, now interpreting.

+--------+-----+
|language|count|
+--------+-----+
|English |1    |
+--------+-----+
```

Fig 10.3: Most common language in the dataset

10.2.3 Average Length of Book Titles

To compute the average length of book titles, the character length of each extracted title was calculated using Spark's length() function. The average of these values was then computed using the avg() aggregation function.

This metric provides an estimate of how descriptive or concise book titles are across the dataset.



```
scala> metadata_df
res64: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 4 more fields]

scala>   .filter(col("title") =!= "")
res65: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [text: string, file_name: string ...

scala>   .select(avg(length(col("title"))).alias("avg_title_length"))
res66: org.apache.spark.sql.DataFrame = [avg_title_length: double]

scala>   .show(false)
+----------------+
|avg_title_length|
+----------------+
|51.0            |
+----------------+
```

Fig 10.4: Average length of book titles (in characters)

10.3 Explanation of Regular Expressions Used

Regular expressions were used to extract metadata from the unstructured text of each book. The following patterns were applied:

Title:

```
  (?i)title:\s*(.*)
```
This pattern extracts the text following the keyword “Title:”, ignoring case sensitivity.

Release Date:

```
  (?i)release date:\s*(.*)
```
This captures the release date information from the book header.

Language:

```
(?i)language:\s*(.*)
```
This extracts the language in which the book is written.

Character Set Encoding:

```
(?i)character set encoding:\s*(.*)
```
This retrieves the encoding used for the text file.

The (?i) flag ensures case-insesitive matching, which is necessary because metadata labels may vary in capitalization across files.

10.4 Challenges and Limitations of Using Regular Expressions

One of the main challenges of using regular expressions for metadata extraction is the lack of strict consistency in text formatting across different books. Some files may have missing metadata fields, variations in spelling, additional whitespace, or metadata appearing in unexpected locations.

Additionally, regular expressions may fail when metadata spans multiple lines or when multiple similar patterns exist within the text, leading to incomplete or incorrect extraction.

10.5 Potential Issues and Real-World Handling

Potential issues with the extracted metadata include missing values, inconsistent date formats, ambiguous language labels, and incorrect matches due to formatting differences. In a real-world scenario, these issues can be addressed by applying data validation rules, normalization techniques, fallback extraction logic, or by combining regex-based extraction with structured metadata sources when available.

Advanced approaches such as natural language processing or schema-based metadata ingestion could further improve extraction accuracy.

Question 11: TF-IDF and Book Similarity using Apache Spark

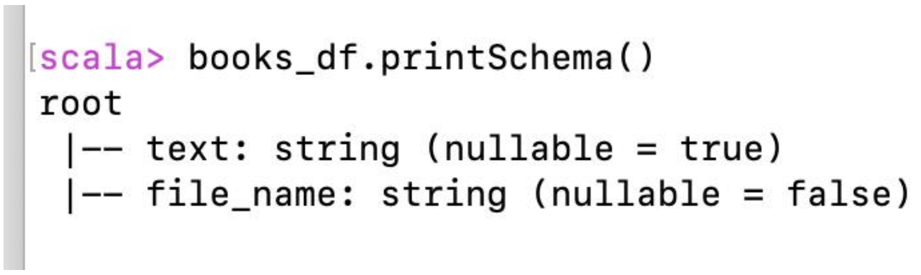
Objective:

The objective of this task is to compute TF-IDF (Term Frequency–Inverse Document Frequency) scores for words in each book and use these scores to measure similarity between books using cosine similarity. This helps in identifying books with similar textual content.

11.1 Dataset Loading

The Project Gutenberg dataset containing multiple .txt book files was loaded into Apache Spark as a DataFrame named books_df. Each row represents a book, consisting of:

- file_name: Name of the book file
- text: Full textual content of the book



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

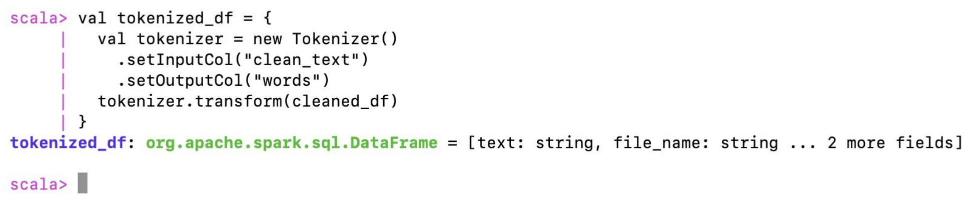
Fig 11.1: Schema of books_df showing file_name and raw text content

11.2 Text Preprocessing

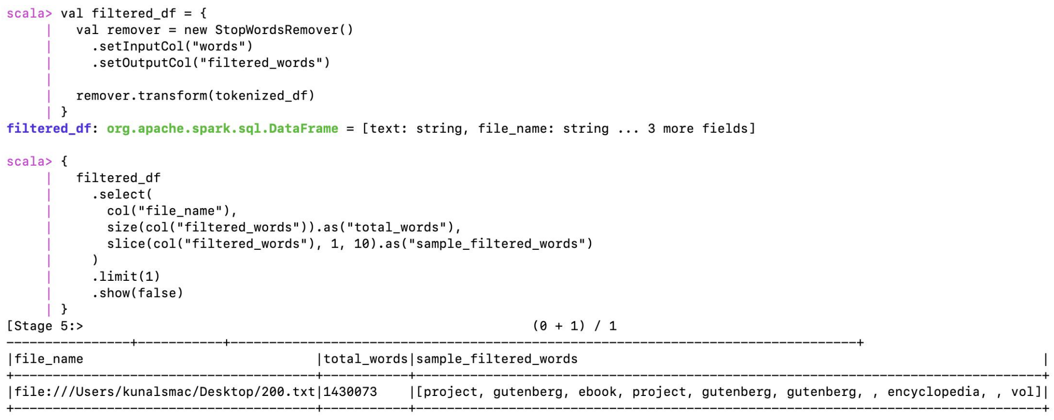
The Project Gutenberg dataset containing multiple .txt book files was loaded into Apache Spark as a DataFrame named books_df. Each row represents a book, consisting of:

1. Removal of Project Gutenberg headers and footers
2. Conversion of text to lowercase
3. Removal of punctuation and special characters
4. Tokenization of text into individual words
5. Removal of stop words using Spark's StopWordsRemover

This preprocessing ensures that only meaningful terms contribute to the TF-IDF computation.



```
scala> val tokenized_df = {
     |   val tokenizer = new Tokenizer()
     |     .setInputCol("clean_text")
     |     .setOutputCol("words")
     |   tokenizer.transform(cleaned_df)
     | }
tokenized_df: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 2 more fields]

scala> 
```



```
scala> val filtered_df = {
     |   val remover = new StopWordsRemover()
     |     .setInputCol("words")
     |     .setOutputCol("filtered_words")
     |
     |   remover.transform(tokenized_df)
     | }
filtered_df: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 3 more fields]

scala> {
     |   filtered_df
     |     .select(
     |       col("file_name"),
     |       size(col("filtered_words")).as("total_words"),
     |       slice(col("filtered_words"), 1, 10).as("sample_filtered_words")
     |     )
     |     .limit(1)
     |     .show(false)
     | }
[Stage 5:>                                                          (0 + 1) / 1
----------------+-----------+----------------------------------------------------------------------------------+
|file_name                              |total_words|sample_filtered_words                                                            |
+---------------------------------------+-----------+---------------------------------------------------------------------------------+
|file:///Users/kunalsmac/Desktop/200.txt|1430073    |[project, gutenberg, ebook, project, gutenberg, gutenberg, , encyclopedia, , vol]|
+---------------------------------------+-----------+---------------------------------------------------------------------------------+
```

Fig 11.2: Sample output after tokenization and stop-word removal

11.3 TF-IDF Computation

Term Frequency (TF):

Term Frequency measures how often a word appears in a document.
It was computed using Spark's CountVectorizer, which converts tokenized words into numerical feature vectors.

Inverse Document Frequency (IDF)

IDF measures how important a word is across the entire corpus. Words appearing in many documents receive lower weight.

TF-IDF

TF-IDF is computed as:

$\text{TF-IDF} = \text{TF} \times \text{IDF}$

This gives higher importance to words that are frequent in a document but rare across the dataset.

```
scala> val idf = new IDF()
idf: org.apache.spark.ml.feature.IDF = idf_268237bc5ac1

scala>   .setInputCol("tf_features")
res92: idf.type = idf_268237bc5ac1

scala>   .setOutputCol("tfidf_features")
res93: res92.type = idf_268237bc5ac1

scala>

scala> val idf_model = idf.fit(tf_df)
idf_model: org.apache.spark.ml.feature.IDFModel = IDFModel: uid=idf_268237bc5ac1, numDocs=1, numFeatures=61877

scala>

scala> val tfidf_df = idf_model.transform(tf_df)
tfidf_df: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 5 more fields]

scala> tfidf_df
res94: org.apache.spark.sql.DataFrame = [text: string, file_name: string ... 5 more fields]

scala>   .select(
     |     col("file_name"),
     |     col("tfidf_features").cast("string").substr(1, 120).as("tfidf_sample")
     |   )
res95: org.apache.spark.sql.DataFrame = [file_name: string, tfidf_sample: string]

scala>   .show(1, false)
26/02/13 01:25:59 WARN DAGScheduler: Broadcasting large task binary with size 1667.9 KiB
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+
|file_name                              |tfidf_sample                                                                                                            |
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+
|file:///Users/kunalsmac/Desktop/200.txt|(61877,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40|
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+

scala>   .show(1, false)
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+
|file_name                              |tf_features_sample                                                                                                      |
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+
|file:///Users/kunalsmac/Desktop/200.txt|(61877,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40|
+---------------------------------------+------------------------------------------------------------------------------------------------------------------------+
```

Fig 11.3: Schema showing tf_featurs and tfidf_features columns

11.4 Book Similarity using Cosine Similarity

Each book was represented as a **TF-IDF vector**.
Cosine similarity was then used to measure similarity between books.

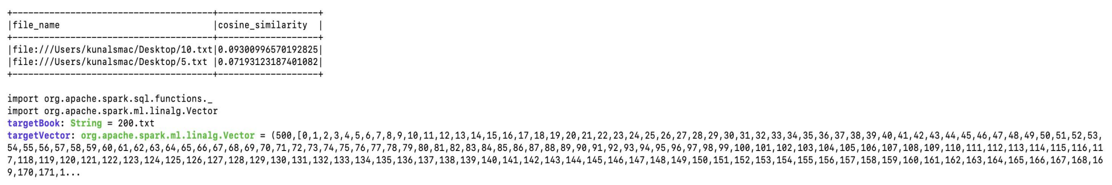
The cosine similarity between two vectors **A** and **B** is defined as:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

- A value close to 1 indicates high similarity
- A value close to 0 indicates low similarity

Using 200.txt as the target book, cosine similarity was calculated with other books in the dataset, and the **top 5 most similar books** were identified.

```
scala> def cosineSimilarity(v1: Vector, v2: Vector): Double = {
     |   val dotProduct = v1.toArray.zip(v2.toArray).map { case (a, b) => a * b }.sum
     |   val norm1 = math.sqrt(v1.toArray.map(x => x * x).sum)
     |   val norm2 = math.sqrt(v2.toArray.map(x => x * x).sum)
     |   if (norm1 == 0.0 || norm2 == 0.0) 0.0 else dotProduct / (norm1 * norm2)
     | }
cosineSimilarity: (v1: org.apache.spark.ml.linalg.Vector, v2: org.apache.spark.ml.linalg.Vector)Double
```



```
+--------------------------------------+-------------------+
|file_name                             |cosine_similarity  |
+--------------------------------------+-------------------+
|file:///Users/kunalsmac/Desktop/10.txt|0.09300996570192825|
|file:///Users/kunalsmac/Desktop/5.txt |0.07193123187401082|
+--------------------------------------+-------------------+

import org.apache.spark.sql.functions._
import org.apache.spark.ml.linalg.Vector
targetBook: String = 200.txt
targetVector: org.apache.spark.ml.linalg.Vector = (500,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,
54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,11
7,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,16
9,170,171,1...
```

Conceptual Discussion: TF-IDF, Cosine Similarity, and Scalability in Large-Scale Text Processing

TF-IDF is useful in text analysis because it reduces the influence of very common words while emphasizing terms that are more distinctive to a specific document, which in turn improves the accuracy of document comparison. Cosine similarity is an appropriate measure for comparing documents since it is independent of document length, works efficiently with high-dimensional sparse vectors, and is widely used in text similarity tasks. However, computing similarity at scale presents challenges, as pairwise comparisons grow quadratically and lead to rapidly increasing memory and computational costs. Apache Spark addresses these issues by enabling distributed computation through RDDs and DataFrames, supporting parallel processing across multiple nodes, and efficiently handling large-scale datasets.

Conclusion:

This experiment successfully demonstrated how Spark can be used to compute TF-IDF scores and identify similar books using cosine similarity. The approach scales well for large text datasets and forms the basis for recommendation systems and text analytics.

Question 12: Author Influence Network

Answer:

Objective:

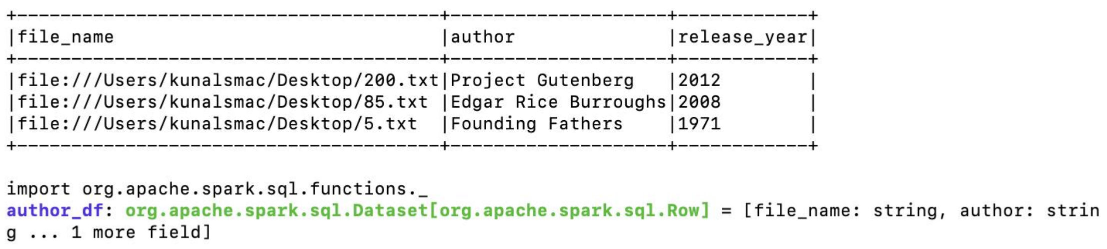
The objective of this task is to construct a **simplified author influence network** based on the **temporal proximity of book release dates**. This network approximates influence relationships between authors.

12.1 Metadata Extraction

From each book's text, the following metadata was extracted using regular expressions:

- Author
- Release Date

The release year was parsed from the release date and converted into a numerical format for comparison.



```
+--------------------------------------+--------------------+------------+
|file_name                             |author              |release_year|
+--------------------------------------+--------------------+------------+
|file:///Users/kunalsmac/Desktop/200.txt|Project Gutenberg   |2012        |
|file:///Users/kunalsmac/Desktop/85.txt |Edgar Rice Burroughs|2008        |
|file:///Users/kunalsmac/Desktop/5.txt  |Founding Fathers    |1971        |
+--------------------------------------+--------------------+------------+

import org.apache.spark.sql.functions._
author_df: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [file_name: string, author: strin
g ... 1 more field]
```

Fig 12.1: Schema showoing extracted author and release year

12.2 Influence Network Construction

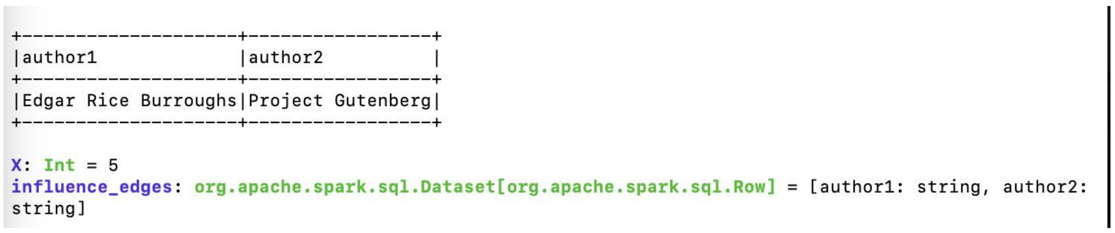
An influence relationship was defined as follows:

Author A influences Author B if Author B released a book within X years of Author A's release

For this experminet:

- The time windows X = 5 years

Pairs of authros satisfying this condition were treated as directed edges:
(author1 --> author2)
These edges were stored in a Spark DataFrame, representing a directed influence network.



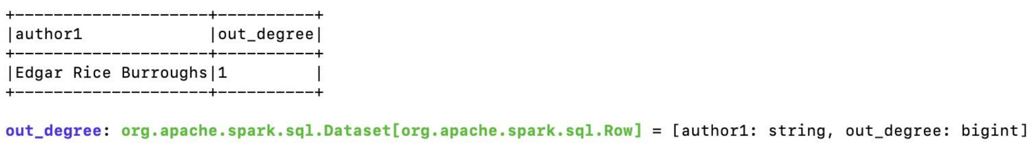
```
+--------------------+-----------------+
|author1             |author2          |
+--------------------+-----------------+
|Edgar Rice Burroughs|Project Gutenberg|
+--------------------+-----------------+

X: Int = 5
influence_edges: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [author1: string, author2:
string]
```

12.3 Network Analysis:

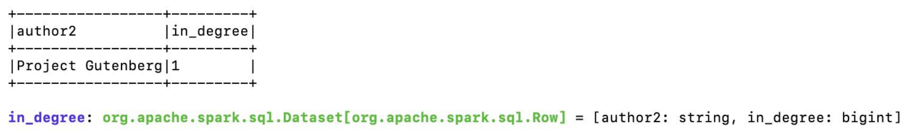
Using the constructed influence network:

- out-degree: Number of authors potentially influenced by an author
- In-degree: Number of authors that potentially influenced an author

The top 5 authors with the highest in-degree and out-degree were identified.



```
+--------------------+----------+
|author1             |out_degree|
+--------------------+----------+
|Edgar Rice Burroughs|1         |
+--------------------+----------+

out_degree: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [author1: string, out_degree: bigint]
```



```
+-----------------+---------+
|author2          |in_degree|
+-----------------+---------+
|Project Gutenberg|1        |
+-----------------+---------+

in_degree: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [author2: string, in_degree: bigint]
```

Fig 12.3: Top authors by in-degree and out-degree

Conceptual Discussion: Representation Choice, Time Window Effects, and Scalability

A DataFrame-based edge list representation was used due to its simplicity, support for easy aggregation using groupBy operations, optimized execution in Spark, and good scalability for large datasets, though it offers limited expressiveness compared to dedicated graph libraries. The choice of the time window (X) significantly impacts the network structure, where a smaller X results in a sparse network, while a larger X produces a denser network but also increases the risk of false positives in inferred influence. The approach has several limitations, as influence is inferred solely from release dates without considering citations, themes, popularity, or handling multiple authors per book. From a scalability perspective, pairwise comparisons grow rapidly with data size, but Spark mitigates this through parallel joins and aggregations, with further optimizations possible using broadcast joins and graph processing frameworks such as GraphFrames and GraphX.

Conclusion:

This task demonstrated how Spark can be used to perform **graph-like analysis** on large text datasets. While simplified, the approach effectively illustrates temporal influence modeling and scalable network analysis.