ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING – SITE

Department of Artificial Intelligence

Course: Distributed Computing for AI ITSC-2112

Assignment A5

Spark: word count

Submitted By: Mintesnot Fikir ID GSR/1669/15

Section: Regular

Submitted to: Dr. Beakal Gizachew

Submission date: January, 2024

1. Introduction

In this assignment, I built upon my previous experiences on hadoop which is the 4th assignment, to exploring Spark, specifically PySpark, for distributed data computing and storage based on text data. This task was conducted within a Jupyter Notebook environment on my personal computer, highlighting the practical application of PySpark in processing and analyzing large datasets. The assignment was given as two main parts: conducting a **Basic Word Count on a single book** and **Extended Word Count**: extending analysis to include a collection of 10 books, all downloaded from Project Gutenberg. This assignment introduced me to spark and hadoop for the first time and also into the bigdata and analysis, with a particular focus on the impact of stopwords on the datasets.

For the **Basic Word Count task,** I implemented PySpark code to analyze word occurrences in a single text, ensuring the analysis was insensitive to case and punctuation, and capable of filtering out stopwords as we instructed from the assignment. Using this initial analysis, compered the effects of including versus excluding **stopwords** on the word count results. An important aspect of this task was utilizing Spark's **DAG Visualization** through the Spark UI, which provided insights into the data processing stages, showcasing the efficiency and capabilities of Spark in handling text data processing.

The second task was implementing **Extended Word Count,** expanding on analyzing 10 books from Project Gutenberg, applying the same criteria for word count analysis across a larger dataset. This extended analysis underscored the scalability of Spark, allowing me to aggregate and analyze data across multiple texts efficiently. By **comparing word counts with and without stopwords**, I gained deeper insights into the commonalities and unique aspects of the text collection. This comprehensive analysis, supported by visual evidence from the **DAG visualization** in Spark's WebUI, not only confirmed the findings from the individual book analysis but also highlighted **Spark's robustness and flexibility** in managing and processing large-scale text data. And finally, I have compered the result from hadoop and spark, and got the same result as expected. I furthered my understanding of distributed computing and the practical applications of Spark in real-world data analysis scenarios.

2. Project Setup

The setup for this project:

- **Apache Spark and PySpark Version 3.3.4**: Served as the backbone for processing large-scale data, enabling efficient analysis and manipulation of text data and other.
- **Java JDK Version 8u202(1.8.0_202)**: Provided the necessary runtime environment for Spark, ensuring smooth execution of Spark applications.
- **Python Version 3.11.5**: Enabled scripting and data manipulation in PySpark, offering the latest features for performance and code readability.
- **Apache Hadoop Version 2.7**: Supported Spark's data handling capabilities, particularly for storing and managing large datasets.

3. Implementation Details

The implementation of the word count analysis for this assignment was structured around two primary tasks: a Basic Word Count analysis on a single text and an Extended Word Count analysis across multiple texts. The objective was to meticulously count word occurrences, applying specific criteria to ensure accuracy and relevance in the results. This section delves into the technical specifics and methodologies employed to achieve these objectives using PySpark within a Jupyter Notebook environment.

To tackle the Basic and Extended Word Count tasks, I started by setting up my development environment with PySpark and essential libraries like Pandas, NLTK, and Pyngrok in a Jupyter Notebook on my PC. I initialized a SparkSession to run locally, specifying an application name "Spark UI" and configuring Spark to use a specific UI port, which Pyngrok then exposed through a public URL. This setup was crucial for accessing Spark's UI remotely, allowing for real-time monitoring and debugging of the application's execution. With these preparations, my environment was ready for the implementation phases of the word count analyses, utilizing PySpark's distributed computing power to process text data from Project Gutenberg effectively.

Initialize SparkContext and set Spark UI access point

```
In [5]: spark = SparkSession.builder\
            .master("local[*]")\
            .appName("Spark UI")\
            .config("spark.ui.port", "4050")\
            .getOrCreate()

        NGROK_TOKEN = "2cMzS23dCc40jCub1EmWsC7lLYF_6SvSu354ptk7Ri79vwJH"
        ngrok.set_auth_token(NGROK_TOKEN)

        ngrok_tunnel = ngrok.connect(4050, bind_tls=True)
        print("Spark UI:", ngrok_tunnel.public_url)

        Spark UI: https://bc01-35-230-164-173.ngrok-free.app

In [6]: sc = spark.sparkContext

In [7]: sc

Out[7]: SparkContext

        Spark UI
        Version
        v3.5.0
        Master
        local[*]
        AppName
        Spark UI
```

Data Preparation and Preprocessing (All Data processing task included in spark implementation)
The analysis commenced with the selection of textual data from Project Gutenberg, ensuring a diverse and substantial dataset for analysis. Each text file underwent preprocessing to normalize the data, which included based on the instruction from the assignment:
- **Case Insensitivity**: All text was converted to lowercase using Python's **lower** () function, ensuring uniformity in word recognition.
- **Punctuation and Numbers Removal**: Utilizing Python's **re** library, all punctuation was stripped from the text, leaving only alphabetic characters and whitespace. This step was crucial for focusing the analysis on words devoid of non-textual elements.
- **Stop Words Filtering**: A list of stop words was prepared and broadcasted using Spark's broadcast mechanism. These stop words were then filtered out from the text using Spark's **filter** () transformation, allowing the focus to shift to more meaningful words in the analysis.

4. Dataset (Books)

For this assignment, I selected ten diverse books from Project Gutenberg, ensuring each text file exceeded the minimum size requirement of 100kB to provide a rich dataset for analysis. The books vary in genre, length, and historical period, offering a broad spectrum for our word count examination. Below is a table detailing the names, file sizes, and character counts of the downloaded books:

Name of the Book	File Name (.txt format)	Size (KB)	Number of Characters
A Visit to the Roman Catacombs	A_visit_to_the_Roman_catacombs_pg71927	257	256,593
Elements of Metaphysics	elements_of_meta_physics_pg71885	1,182	1,181,041
History For Ready Reference	history_For_Ready_Reference_pg71897	5,159	5,150,599
Little Women	Little_Women_pg37106	648	648,914
Middlemarch	Middlemarch_pg5197	1,823	1,799,413
The Australian Aboriginal	The_Australian_aboriginal_pg71940	840	835,784
The Green Hat	The_Green_hat_pg71913	529	517,486
The Iliad	The_Iliad_pg6130	223	220,426
The Romance of Lust	The_Romance_of_Lust_pg30254	234	233,569
The Tempest	The_Tempest_pg23042	102	101,171

Table 1: Pre-Processed book data (.txt format)

Project Gutenberg is a library of over 70,000 free eBooks: https://www.gutenberg.org/

5. Part 1 A: Basic Word Count for a Single Text File

Basic Word Count (From a single book)

The first task of the assignment was to implement a Basic Word Count analysis on a single text file, specifically book called "Little Women" which is downloaded from Project Gutenberg. This task was designed to not only count the occurrences of each word within the text but also to apply a series of filters to ensure the analysis was both meaningful and in accordance with the assignment's specifications. Below, I detail the step-by-step approach I took to accomplish this task, utilizing PySpark within a Jupyter Notebook environment on my personal computer.

So, the first step is, began with loading the text file into a Spark Resilient Distributed Dataset (RDD) using PySpark's text File method. This step ensured that the text was ready for distributed processing:

Read single book

```
In [3]: readsinglebook = sc.textFile("Little_Women.txt")
```

Then Load and Broadcast Stopwords into PySpark: To refine the word count by removing common stopwords, I loaded a predefined list of stopwords from a text file. These stopwords were then broadcasted across the cluster to ensure efficient access by all nodes during the filtering process:

Load and broadcast stopwords

```
In [9]: import re
        stopwords_path = "stopwords.txt"
        with open(stopwords_path, 'r') as file:
            stopwords = set(file.read().split())
        bcast = sc.broadcast(stopwords)
```

Remove Punctuation and Numbers: A custom function, $rmsplitpunon$, was defined to remove all punctuation and numbers from the text, utilizing Python's $re.sub$ for regex-based substitution. This function was crucial for ensuring that only alphabetic characters were considered during the word count:

Remove panctuation

```
In [10]: def rmsplitpunon(line):
             line = re.sub(r'[^a-zA-Z\s]', '', line)
             return line.split()
```

5.1. Word Count implementation

As the assignment instruction asks, the core of this task is not only counting the occurrence of each word but also other data processing involved performing the word count with several RDD transformations:

- *FlatMap*: Applied the **rmsplitpunon** function to split the text into words, removing punctuation and numbers.

- *Filter*: Excluded words present in the broadcasted list of stopwords, focusing the analysis on more meaningful words.

- *Map*: Converted words to lowercase (for case insensitivity) and mapped each word to a tuple containing the word and the count **1**.

- *ReduceByKey*: Aggregated the counts for each word across the dataset.

- *SortBy*: Sorted the results by count in descending order to identify the most frequent words.

Perform Word Count

- Word count with filtering stopwords, and also removing punctuation, and numbers

Based on the assignment instractun in addition to basic word counting my code also do the following:

1. It must be case insensitive (see lower() in Python)
2. It must ignore all punctuation (see, for example, translate() in Python)
3. It must ignore stop words (see filter() in Spark)
4. The output must be sorted by count in descending order (see sortBy() in Spark)

```
In [11]: wordcountsingle = readsinglebook.flatMap(lambda line: rmsplitpunon(line)) \
                               .filter(lambda word: word.lower() not in bcast.value) \
                               .map(lambda word: (word.lower(), 1)) \
                               .reduceByKey(lambda x, y: x + y) \
                               .sortBy(lambda x: x[1], ascending=False)
```

The program implemented here performs an analysis on a text file by counting the occurrences of each word. It generates an output in the form of a text file that presents a list containing each word along with the corresponding frequency of its occurrences. Usage by treating variations of a word as the same entity, regardless of case or surrounding punctuation.

Saving Results: The results were saved in two formats to ensure both compatibility with Spark's default storage mechanism and accessibility for further analysis:

Default RDD Format: Saved directly using Spark's saveAsTextFile method.

Single Text File: To facilitate easier review, the RDD was coalesced into a single partition and then saved as a .txt file, which was subsequently moved to the specified directory.

Saving in Default RDD Format and also as a text

```
In [14]:  savedir = "/outputs/wordcountsinglebook"

          wordcountsingle.saveAsTextFile(savedir)

          temp_output_dir = "/output/wordcountsinglebook"
          wordcountsingle.coalesce(1).saveAsTextFile(temp_output_dir)
          !mv $(find /content/wordcountsinglebook -type f -name 'part-00000') '/outputs/wordcountsinglebook.txt'
          !rm -rf /content/wordcountsinglebook
```

Display Top Words: Finally, the top 25 most frequent words were extracted from the RDD and displayed using a Pandas DataFrame for a clear and structured presentation of the results:

Display Top Words

```
In [15]: top25 = wordcountsingle.take(25)
         dfcountsingle = pd.DataFrame(top25, columns=['Word', 'Count Without Stopwords'])
         dfcountsingle.index = range(1, len(dfcountsingle) + 1)
         dfcountsingle
```

Out[15]:

	Word	Count Without Stopwords
1	jo	841
2	meg	585
3	amy	385
4	dont	375
5	laurie	349
6	beth	315
7	mother	253
8	march	250
9	good	241
10	girls	217
11	time	195
12	asked	189
13	young	173
14	face	170
15	ill	165
16	dear	162
17	day	156
18	illustration	143
19	looked	137
20	eyes	131
21	cried	128
22	great	124
23	head	121
24	didnt	119
25	wont	117

5.2. Let's compere without stop-word and with stop-word included

In the first task of our assignment, I focused on performing a word occurrence count on a single text file, "Little Women," extracted from Project Gutenberg. This analysis was conducted twice: once including stopwords in our dataset and once filtering them out. This dual approach allowed us to examine the impact of stopwords on the word count results, offering a better understanding of their role in textual analysis. The implementation details and comparative analysis are discussed below.

Performing Word Count **with Stopwords included** vs **Without Stopwords**

Two RDD pipelines were constructed:

- **With Stopwords**: This pipeline processed the text without filtering out stopwords, directly counting the occurrences of all words post-punctuation removal.

- **Without Stopwords**: This pipeline included an additional filtering step to remove stopwords, thus focusing the word count on more content-rich words.

Both pipelines implemented case-insensitive counting, ignored punctuation, and sorted the results by word frequency in descending order.

Lets compere without stopword and with stop word

```
In [16]: withsword = readsinglebook.flatMap(lambda line: rmsplitpunon(line)) \
                        .map(lambda word: (word.lower(), 1)) \
                        .reduceByKey(lambda x, y: x + y) \
                        .sortBy(lambda x: x[1], ascending=False)


         withoutsword = readsinglebook.flatMap(lambda line: rmsplitpunon(line)) \
                        .filter(lambda word: word.lower() not in bcast.value) \
                        .map(lambda word: (word.lower(), 1)) \
                        .reduceByKey(lambda x, y: x + y) \
                        .sortBy(lambda x: x[1], ascending=False)
```

Comparison of with stop word and without stop word

```
In [22]: top25without = withoutsword.take(25)
         top25with = withsword.take(25)

         dfwithout = pd.DataFrame(top25without, columns=['Word', 'Count Without Stopwords'])
         dfwith = pd.DataFrame(top25with, columns=['Word', 'Count With Stopwords'])

         dfwithout.index = range(1, len(dfwithout) + 1)
         dfwith.index = range(1, len(dfwith) + 1)

         comparisondf = pd.concat([dfwithout, dfwith], axis=1)
         comparisondf
```

Out[22]:

	Word	Count Without Stopwords	Word	Count With Stopwords
1	jo	841	and	5095
2	meg	585	the	4808
3	amy	385	to	3125

Save the Word Count Results

```
In [18]: dir1 = "/content/drive/MyDrive/MintesnotF-A5Spark-assignment/outputs/compered"

         outwithout1 = f"{dir1}/resultswithoutstopwords"
         outwith1 = f"{dir1}/resultswithstopwords"

         withoutsword.saveAsTextFile(outwithout1)
         withsword.saveAsTextFile(outwith1)
```

5.2.1. Comparative Analysis

	Word	Count Without Stopwords	Word	Count With Stopwords
1	jo	841	and	5095
2	meg	585	the	4808
3	amy	385	to	3125
4	dont	375	a	2833
5	laurie	349	her	2235
6	beth	315	of	2117
7	mother	253	i	1886
8	march	250	in	1505
9	good	241	she	1446
10	girls	217	it	1444
11	time	195	you	1354
12	asked	189	for	1285
13	young	173	was	1268
14	face	170	as	1155
15	ill	165	with	1107
16	dear	162	that	1003
17	day	156	jo	841
18	illustration	143	but	819
19	looked	137	he	791
20	eyes	131	so	688
21	cried	128	at	670
22	great	124	had	668
23	head	121	be	630
24	didnt	119	on	598
25	wont	117	said	589

The comparative analysis of word occurrence counts with and without stopwords revealed significant differences in the most common words. The inclusion of stopwords, as expected, led to a list dominated by common English words such as 'and', 'the', 'to', which, while frequent, offer limited insight into the thematic content of the text. Conversely, filtering out stopwords shifted the focus towards more meaningful words specific to the narrative and characters of "Little Women," such as 'jo', 'meg', 'amy', highlighting their central role in the story.

The side-by-side comparison of the top 25 words from each approach underscores the transformative effect of stopwords on textual analysis. Words like 'jo', 'meg', 'amy' emerge as significant in the absence of stopwords, whereas their importance is overshadowed by common stopwords when these are not filtered out. This contrast not only demonstrates the utility of stopword filtering in uncovering thematic and character-driven elements of a text but also reflects on the nature of language use in literature not on the performance of distributed computing.

6. Part 1 B: Basic Word Count from multiple text file(10 books)

Extended Word Count (From a multiple books)

Following the previous work laid out in the Basic Word Count task for a single book as text file, the next phase of the assignment expanded the scope to encompass a broader dataset - specifically, 10 books downloaded from Project Gutenberg. This ambitious extension aimed to test the scalability of the spark implementation aggregate word occurrence data across a diverse literary collection, offering a comprehensive view of word usage and frequency when considering a larger corpus. The implementation process, guided by the assignment's rigorous criteria, involved several key steps to ensure accurate, insightful analysis.

Data Aggregation and Preprocessing

The initial step involved the aggregation of text data from ten distinct books. These texts, each book downloaded from Project Gutenberg and with size more than 100kb as we instructed from the assignment were combined into a single dataset for analysis. This process utilized the following command to read all text files within the specified directory:

So, the first step is, began with loading the text files into a Spark Resilient Distributed Dataset (RDD) using PySpark's text File method. This step ensured that the text was ready for distributed processing:

Load/Read books and Aggregating Text Data

```
In [5]: bookspath = "/books"
        book_files = [f for f in os.listdir(bookspath) if f.endswith('.txt')]
        book_paths = [os.path.join(bookspath, f) for f in book_files]

        read10books = sc.emptyRDD()
        for book_path in book_paths:
            book_rdd = sc.textFile(book_path)
            read10books = read10books.union(book_rdd) if not read10books.isEmpty() else book_rdd
```

Initially, a set of ten books, each a plain text file exceeding 100 kB, was curated from Project Gutenberg. These books were stored in a designated directory within local storage, setting the stage for distributed processing.

Loading and Broadcasting Stop Words:

Load and broadcast stopwords

```
In [1]: stopwordspath = "stopwords.txt"
        with open(stopwordspath, 'r') as file:
            stopwords = set(file.read().split())
        bcaststopwords = sc.broadcast(stopwords)
```

The text data from ten books were aggregated into a single RDD using *union* (). This approach facilitated the combined analysis of word occurrences across the entire dataset.

A comprehensive list of stopwords was loaded and broadcasted to all worker nodes using Spark's broadcast mechanism. This ensured efficient filtering of stopwords across the distributed dataset.

Remove Punctuation and Numbers: A custom function, $rmsplitpunon$, was defined to remove all punctuation and numbers from the text, utilizing Python's $re.sub$ for regex-based substitution. This function was crucial for ensuring that only alphabetic characters were considered during the word count:

```
In [6]: def rmandsplitpanc(line):
            line = re.sub(r'[^a-zA-Z\s]', '', line)
            return line.split()
```

6.1. Extended Word Count Implementation

As the assignment instruction asks, the core of this task is not only counting the occurrence of each word but also other data processing involved performing the word count with several RDD transformations:

- ***FlatMap***: Applied the ***rmsplitpunon*** function to split the text into words, removing punctuation and numbers.

- ***Filter***: Excluded words present in the broadcasted list of stopwords, focusing the analysis on more meaningful words.

- ***Map***: Converted words to lowercase (for case insensitivity) and mapped each word to a tuple containing the word and the count **1**.

- ***ReduceByKey***: Aggregated the counts for each word across the dataset.

- ***SortBy***: Sorted the results by count in descending order to identify the most frequent words.

Perform Word Count

- Word count with filtering stopwords, and also removing punctuation, and numbers Based on the assignment instractun in addition to basic word counting my code also do the following:

1. It must be case insensitive (see lower() in Python)
2. It must ignore all punctuation (see, for example, translate() in Python)
3. It must ignore stop words (see filter() in Spark)
4. The output must be sorted by count in descending order (see sortBy() in Spark)

```
In [7]: word_counts_with_10books = read10books.flatMap(lambda line: rmandsplitpanc(line)) \
                                .filter(lambda word: word.lower() not in bcaststopwords.value) \
                                .map(lambda word: (word.lower(), 1)) \
                                .reduceByKey(lambda x, y: x + y) \
                                .sortBy(lambda x: x[1], ascending=False)
```

The program implemented here performs an analysis on all text file combined by counting the occurrences of each word. It generates an output in the form of a text file that presents a list containing each word along with the corresponding frequency of its occurrences. Usage by treating variations of a word as the same entity, regardless of case or surrounding punctuation.
Saving Results: The results were saved in two formats to ensure both compatibility with Spark's default storage mechanism and accessibility for further analysis:

Default RDD Format: Saved directly using Spark's saveAsTextFile method.

Single Text File: To facilitate easier review, the RDD was coalesced into a single partition and then saved as a .txt file, which was subsequently moved to the specified directory.

Saving in Default RDD Format and also as a text

```
In [8]:  # as RDD format
         savedir10 = "/content/drive/MyDrive/MintesnotF-A5Spark-assignment/outputs/word_counts_with_10books"

         word_counts_with_10books.saveAsTextFile(savedir10)


         # as a text file
         outdir = "/content/word_counts_with_10books"
         wordcountsingle.coalesce(1).saveAsTextFile(outdir)
         !mv $(find /content/word_counts_with_10books -type f -name 'part-00000') '/content/drive/MyDrive/MintesnotF-A5Spark-assignment/ou
         !rm -rf /content/word_counts_with_10books
```

Display Top Words: Finally, the top 25 most frequent words from all 10 books combined were extracted from the RDD and displayed using a Pandas DataFrame for a clear and structured presentation of the results:

Display Top Words

```
In [25]:  top20word=word_counts_with_10books.take(25)
          top20wordDF = pd.DataFrame(top20word, columns=['Word', '  Count Without Stopwords from all 10 books'])

          top20wordDF.index = range(1, len(top20wordDF) + 1)
          top20wordDF
```

Out[25]:

	Word	Count Without Stopwords from all 10 books
1	government	2723
2	united	2559
3	time	2488
4	great	2161
5	volume	1995
6	people	1610
7	general	1596
8	men	1579
9	man	1576
10	good	1395
11	work	1274
12	life	1175
13	thought	1162
14	years	1148
15	south	1132
16	war	1129
17	long	1099
18	british	1078
19	order	1054
20	day	1036
21	american	980
22	number	977
23	hand	955
24	president	950
25	country	942

5.2. Let's compere without stop-word and with stop-word included on multiple books

In the first task of our assignment, I focused on performing a word occurrence count on a on multiple book text file extracted from Project Gutenberg. But the analysis is based on stop word excluded only in this section I was conducted twice the word count ways: once including stopwords in our dataset and once filtering them out. This dual approach allowed us to examine the impact of stopwords on the word count

10

results, offering a better understanding of their role in textual analysis. The implementation details and comparative analysis are discussed below.

Performing Word Count **with Stopwords included** vs **Without Stopwords**

Two RDD pipelines were constructed:

- **With Stopwords**: This pipeline processed the text without filtering out stopwords, directly counting the occurrences of all words post-punctuation removal.

- **Without Stopwords**: This pipeline included an additional filtering step to remove stopwords, thus focusing the word count on more content-rich words.

Both pipelines implemented case-insensitive counting, ignored punctuation, and sorted the results by word frequency in descending order.

Lets compere without stopword and with stop word

```
[25]: word_counts_with_stopwords = read10books.flatMap(lambda line: rmandsplitpanc(line)) \
                                 .map(lambda word: (word.lower(), 1)) \
                                 .reduceByKey(lambda x, y: x + y) \
                                 .sortBy(lambda x: x[1], ascending=False)

      word_counts_without_stopwords = read10books.flatMap(lambda line: rmandsplitpanc(line)) \
                                    .filter(lambda word: word.lower() not in bcaststopwords.value) \
                                    .map(lambda word: (word.lower(), 1)) \
                                    .reduceByKey(lambda x, y: x + y) \
                                    .sortBy(lambda x: x[1], ascending=False)
```

Display comparison of with stop word and without stop word

```
In [26]: top_25_without = word_counts_without_stopwords.take(25)
         top_25_with = word_counts_with_stopwords.take(25)

         df_without_stopwords = pd.DataFrame(top_25_without, columns=['Word', 'Count Without Stopwords'])
         df_with_stopwords = pd.DataFrame(top_25_with, columns=['Word', 'Count With Stopwords'])

         df_without_stopwords.index = range(1, len(df_without_stopwords) + 1)
         df_with_stopwords.index = range(1, len(df_with_stopwords) + 1)

         comparison_df = pd.concat([df_without_stopwords, df_with_stopwords], axis=1)

         comparison_df
```

Out[26]:

	Word	Count Without Stopwords	Word	Count With Stopwords
1	government	2723	the	128046
2	united	2559	of	79598
3	time	2488	and	52442
4	great	2161	to	49129

Save the Word Count Results

```
In [18]: dir1 = "/content/drive/MyDrive/MintesnotF-A5Spark-assignment/outputs/compered"

         outwithout1 = f"{dir1}/resultswithoutstopwords"
         outwith1 = f"{dir1}/resultswithstopwords"

         withoutsword.saveAsTextFile(outwithout1)
         withsword.saveAsTextFile(outwith1)
```

11

5.2.1. Comparative Analysis

The comparative analysis involved extracting the top 25 most common words from both sets of results and presenting them side by side for a clear juxtaposition:

Out[26]:

	Word	Count Without Stopwords	Word	Count With Stopwords
1	government	2723	the	128046
2	united	2559	of	79598
3	time	2488	and	52442
4	great	2161	to	49129
5	volume	1995	in	36484
6	people	1610	a	35970
7	general	1596	that	20586
8	men	1579	is	16232
9	man	1576	it	16069
10	good	1395	as	15305
11	work	1274	was	14943
12	life	1175	for	13702
13	thought	1162	with	13671
14	years	1148	be	13627
15	south	1132	i	13118
16	war	1129	by	12471
17	long	1099	which	11385
18	british	1078	on	10351
19	order	1054	he	9929
20	day	1036	not	9851
21	american	980	this	9561
22	number	977	at	9555
23	hand	955	his	9111
24	president	950	her	9091
25	country	942	had	8508

The comparative analysis of word occurrences across ten books from Project Gutenberg, both with and without the inclusion of stopwords, yields insightful distinctions that underscore the profound impact of stopwords on textual analysis. On one hand, the list of the most common words excluding stopwords like "government," "united," and "time"—highlights thematic elements and subject matter intrinsic to the corpus. These words, which top the frequency count in the absence of stopwords, suggest a focus on governance, societal structures, and temporal themes within the texts analyzed. This distilled list offers a clearer view into the substantive content of the literature, pointing towards the authors' focal points and the historical, political, and philosophical discussions woven into their narratives.

When stopwords are included in the analysis, the resulting list is dominated by function words such as "*the*," "of," and "*and*." These words, while essential for grammatical structure and coherence, do not contribute significantly to the thematic or semantic understanding of the text. Their overwhelming prevalence evidenced by "the" appearing *128,046* times skews the word frequency distribution towards linguistic connectors rather than content words. This contrast not only highlights the utility of stopwords in linguistic construction but also delineates their role as diluters of thematic focus in broad-scale lexical analyses. Through this comparative lens, the exclusion of stopwords emerges as a pivotal methodological choice in uncovering the thematic essence and narrative priorities within a corpus, offering a more concentrated glimpse into the intellectual and thematic currents that pervade literary works.

6. Analysis

Part 1 Question 1

1. In the PySpark REPL, run your basic word count program on a single text file.

    A. What are the 25 most common words? Include a screenshot of program output to back-up your claim.

    B. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.

Solution:

A. What are the 25 most common words? Include a screenshot of program output to back-up your claim.

Answer: My Top 25 Words from a single book text:

Display Top Words

```
In [15]: top25 = wordcountsingle.take(25)
         dfcountsingle = pd.DataFrame(top25, columns=['Word', 'Count Without Stopwords'])
         dfcountsingle.index = range(1, len(dfcountsingle) + 1)
         dfcountsingle
```

Out[15]:

	Word	Count Without Stopwords
1	jo	841
2	meg	585
3	amy	385
4	dont	375
5	laurie	349
6	beth	315
7	mother	253
8	march	250
9	good	241
10	girls	217
11	time	195
12	asked	189
13	young	173
14	face	170
15	ill	165
16	dear	162
17	day	156
18	illustration	143
19	looked	137
20	eyes	131
21	cried	128
22	great	124
23	head	121
24	didnt	119
25	wont	117

Running the basic word count program on "Little Women" and filtering out stopwords, the analysis provided a clear insight into the narrative's focus through the lens of word frequency. The character 'Jo'

13

stands out significantly with 841 mentions, underscoring her central role in the story. Following closely are 'Meg' with 585 mentions, 'Amy' with 385, and 'Beth' with 315, highlighting the prominence of the March sisters in the novel.

The analysis also sheds light on key themes and interactions within the narrative. For instance, words related to family and interpersonal dynamics such as 'mother' (253 mentions) and 'girls' (217 mentions) were frequently observed. The term 'good' appeared 241 times, suggesting a moral or evaluative aspect pervasive in the novel's discourse. Additionally, 'time' (195 mentions) and 'day' (156 mentions) point towards the passage of time as a significant theme.

B. *How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.*

Answer:

According to the Spark UI visualization we have 7 stages and 2 of them are skipped and only 5 of them are completed stages.

Here are the 3 completed jobs

Here are DAG Visualization for 5 completed stages

Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 4 s
Locality Level Summary: Process local: 2
Input Size / Records: 771.6 KiB / 13817
Shuffle Write Size / Records: 104.0 KiB / 52
Associated Job Ids: 0
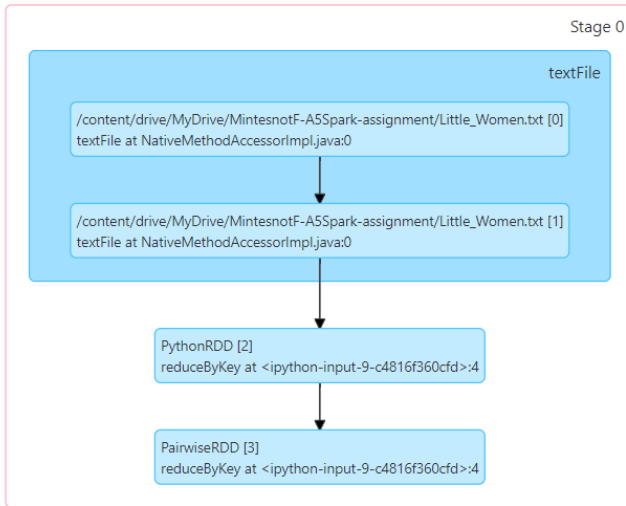
▾ DAG Visualization

Details for Stage 1 (Attempt 0)

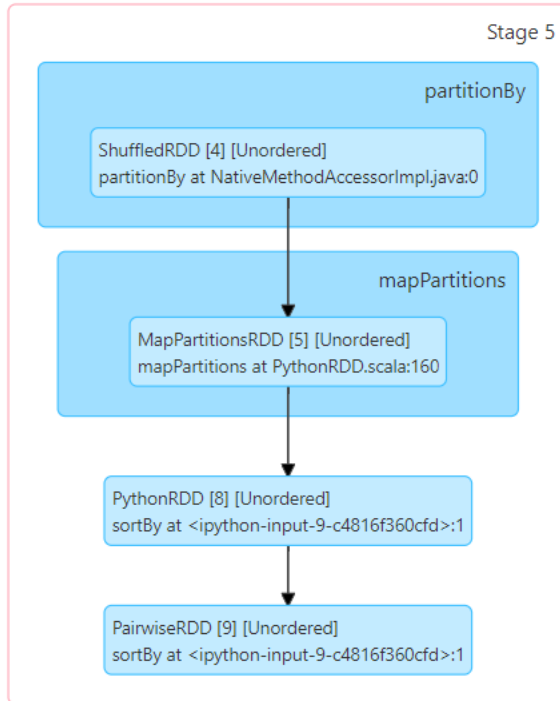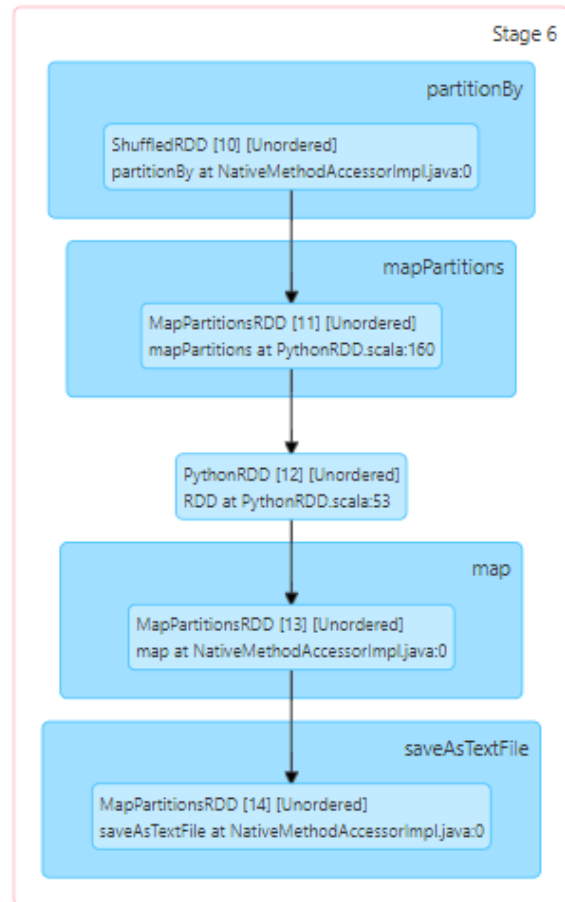**Resource Profile Id:** 0
Total Time Across All Tasks: 0.9 s
Locality Level Summary: Node local: 2
Shuffle Read Size / Records: 104.0 KiB / 52
Associated Job Ids: 0
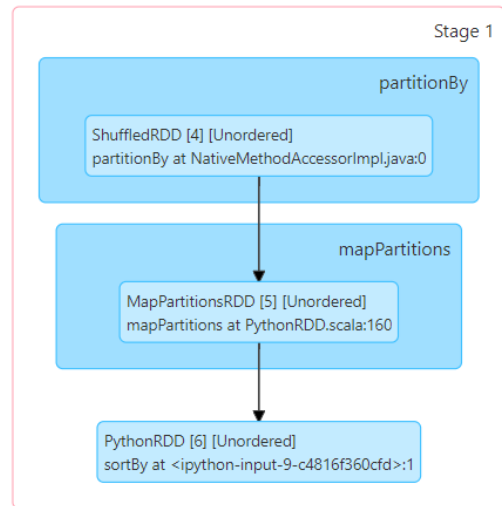
▾ DAG Visualization

Details for Stage 3 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0.4 s
Locality Level Summary: Node local: 2
Shuffle Read Size / Records: 104.0 KiB / 52
Associated Job Ids: 1

▾ DAG Visualization
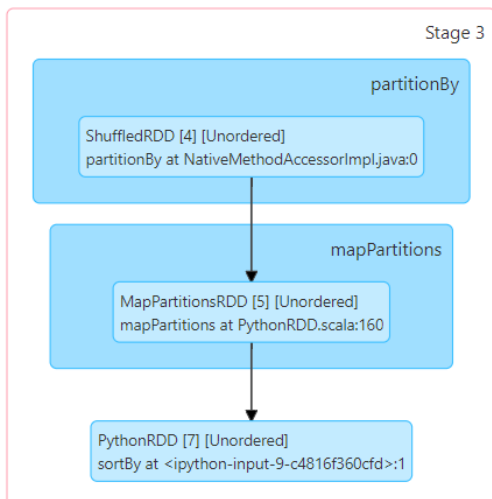
Details for Stage 5 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0.9 s
Locality Level Summary: Node local: 2
Shuffle Read Size / Records: 104.0 KiB / 52
Shuffle Write Size / Records: 84.2 KiB / 48
Associated Job Ids: 2

▼ DAG Visualization

Stage 5

partitionBy

ShuffledRDD [4] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions

MapPartitionsRDD [5] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [8] [Unordered]
sortBy at <ipython-input-9-c4816f360cfd>:1

PairwiseRDD [9] [Unordered]
sortBy at <ipython-input-9-c4816f360cfd>:1

Details for Stage 6 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 2
Output Size / Records: 126.8 KiB / 8416
Shuffle Read Size / Records: 84.2 KiB / 48
Associated Job Ids: 2

▼ DAG Visualization

Stage 6

partitionBy

ShuffledRDD [10] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions

MapPartitionsRDD [11] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [12] [Unordered]
RDD at PythonRDD.scala:53

map

MapPartitionsRDD [13] [Unordered]
map at NativeMethodAccessorImpl.java:0

saveAsTextFile

MapPartitionsRDD [14] [Unordered]
saveAsTextFile at NativeMethodAccessorImpl.java:0

16

Here are DAG Visualization for 3 of them completed jobes

Details for Job 0

Status: SUCCEEDED
Submitted: 2024/02/15 10:23:27
Duration: 3 s
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization
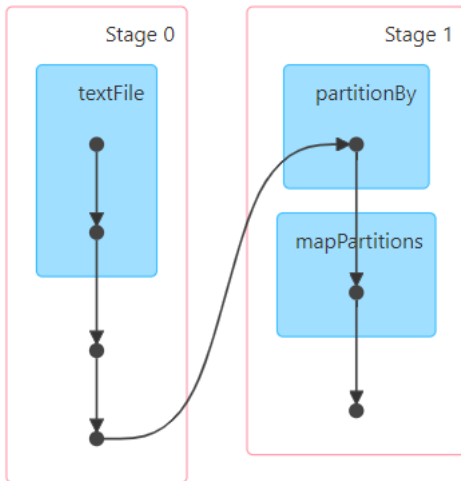
Details for Job 1
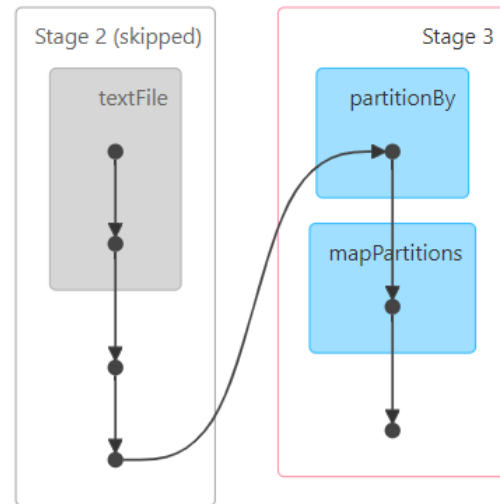
Status: SUCCEEDED
Submitted: 2024/02/15 10:23:30
Duration: 0.3 s
Completed Stages: 1
Skipped Stages: 1

▸ Event Timeline
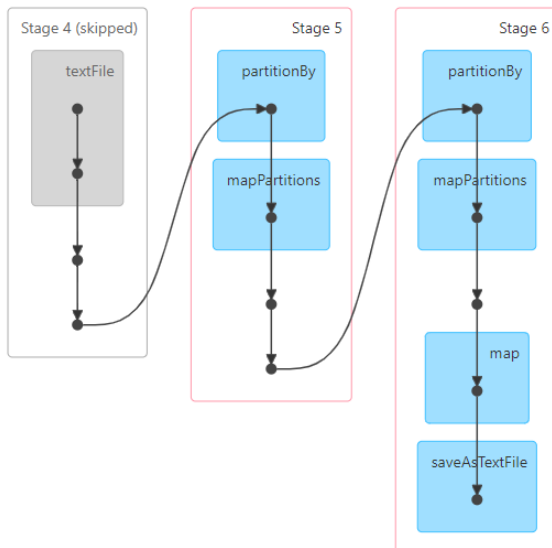▾ DAG Visualization

Details for Job 2

Status: SUCCEEDED
Submitted: 2024/02/15 10:24:16
Duration: 1 s
Completed Stages: 2
Skipped Stages: 1

▸ Event Timeline
▾ DAG Visualization

The Execution Is Broken Down into 7 Stages, here is the analysis The Above DAG Visualization Result

In the execution of my basic word count program on a single text file using PySpark, the Spark UI revealed that the job was broken up into seven stages, with two of those stages being skipped. Specifically, the completed stages were 0, 1, 3, 5, and 6, indicating a detailed process that Spark undertook to achieve the word count, while stages 2 and 4 were skipped due to optimizations made by Spark's execution plan or because their outputs were already available from previous computations.

Explanation of Stages

- **Stage 0** focused on reading the text file and applying a **reduceByKey** operation, which necessitated shuffling data to aggregate word counts. This foundational stage set the groundwork for subsequent transformations.

- **Stage 1 and 3** involved sorting operations (**sortBy**) that likely required shuffling for ordering the word counts in descending order. These stages are critical for ensuring the final output is sorted as per the assignment's requirements.

- **Stage 5 and 6** further involved sorting and preparing the data for output, with stage 6 specifically handling the **saveAsTextFile** operation to write the sorted word counts to the filesystem.

The skipped stages (2 and 4) indicate that Spark optimized the execution plan by eliminating redundant or unnecessary computations. Spark's ability to dynamically adjust the execution plan based on data dependencies and the current state of computations helps in optimizing the overall processing time and resource utilization.

Spark's Execution Plan: The DAG (Directed Acyclic Graph) visualization provided by Spark's WebUI is instrumental in understanding the execution plan and the stages involved. Each stage represents a set of transformations that can be computed independently, with shuffles marking the boundaries between stages. The need for stages arises from operations like reduceByKey and sortBy that require data to be shuffled across partitions. The optimization leading to skipped stages is a testament to Spark's efficiency in managing computations, where it avoids redoing work or skips stages that don't contribute to the final outcome.

Part 1 Question 2

2. In the PySpark REPL, run your extended word count program on all 10 text files.

    A. What are the 25 most common words? Include a screenshot of program output to back-up your claim.

    B. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.

Solution:

A. What are the 25 most common words? Include a screenshot of program output to back-up your claim.

Answer: My Top 25 Words from a single book text:

Display Top Words

```
In [24]: top20word=word_counts_with_10books.take(25)
         top20wordDF = pd.DataFrame(top20word, columns=['Word', '  Count Without Stopwords from all 10 books'])

         top20wordDF.index = range(1, len(top20wordDF) + 1)
         top20wordDF
```

Out[24]:

	Word	Count Without Stopwords from all 10 books
1	government	2723
2	united	2559
3	time	2488
4	great	2161
5	volume	1995
6	people	1610
7	general	1596
8	men	1579
9	man	1576
10	good	1395
11	work	1274
12	life	1175
13	thought	1162
14	years	1148
15	south	1132
16	war	1129
17	long	1099
18	british	1078
19	order	1054
20	day	1036
21	american	980
22	number	977
23	hand	955
24	president	950
25	country	942
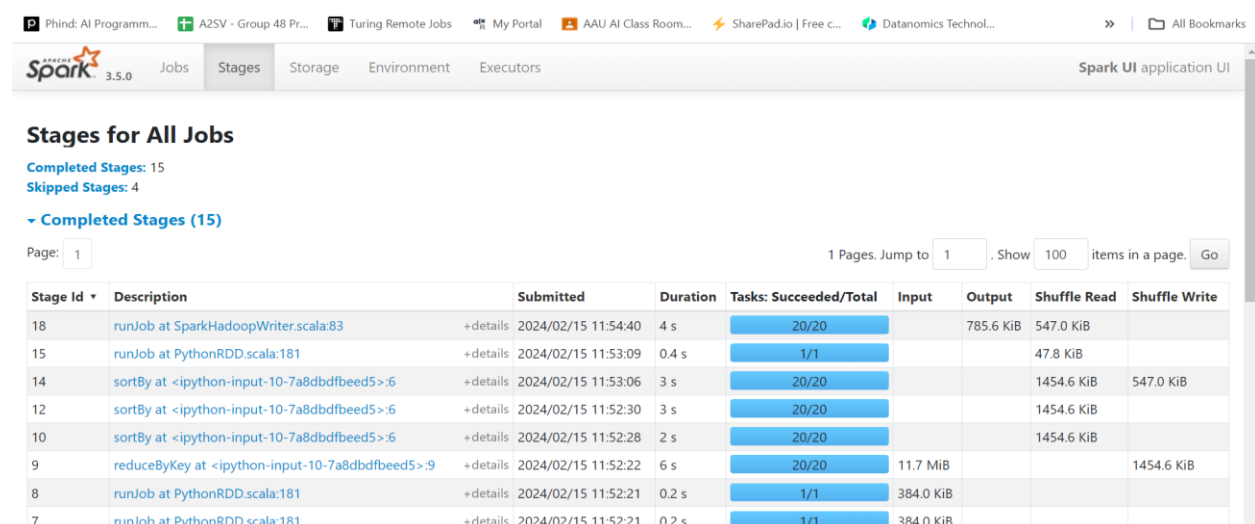
Analyzing the word counts from 10 books after filtering out stopwords revealed significant themes such as governance ("government"), unity ("united"), and the passage of time ("time"), with these words appearing most frequently. This suggests the books delve into historical and political discussions, reflecting on societal structures and human endeavors. Other frequent words like "great," "people," and "life" hint at discussions on grand ideas, societal dynamics, and human experiences. Terms related to historical narratives or analyses, such as "war," "*british*," and "*american*," indicate a focus on military conflicts, national identity, and leadership. This concise analysis, leveraging PySpark for data processing, underscores the thematic richness of the texts and offers insights into collective human conditions and societal observations, demonstrating the effectiveness of big data tools in literary exploration.

1 government	2723	10 good 1395		19 order 1054	
2 united2559		11 work 1274		20 day 1036	
3 time 2488		12 life 1175		21 american 980	
4 great 2161		13 thought 1162		22 number 977	
5 volume 1995		14 years 1148		23 hand 955	
6 people 1610		15 south 1132		24 president 950	
7 general 1596		16 war 1129		25 country 942	
8 men 1579		17 long 1099			
9 man 1576		18 british 1078			

B. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.

Answer:

According to the Spark UI visualization we have 19 stages and 4 of them are skipped and only 15 of them are completed stages.

6	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:21	0.2 s	1/1	384.0 KiB	
5	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:20	0.2 s	1/1	384.0 KiB	
4	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:20	0.3 s	1/1	384.0 KiB	
3	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:19	0.3 s	1/1	384.0 KiB	
2	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:19	0.4 s	1/1	384.0 KiB	
1	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:18	0.3 s	1/1	384.0 KiB	
0	runJob at PythonRDD.scala:181	+details 2024/02/15 11:52:16	1 s	1/1	384.0 KiB	

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

▼ Skipped Stages (4)
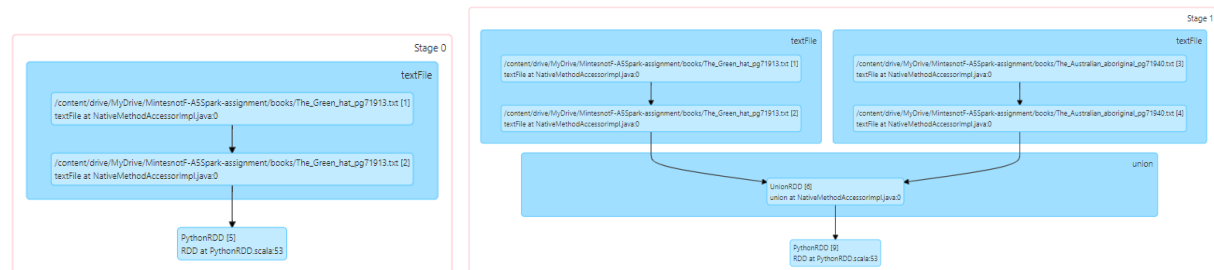
Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▼	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	sortBy at <ipython-input-10-7a8dbdfbeed5>:6	+details	Unknown	Unknown	0/20				
16	reduceByKey at <ipython-input-10-7a8dbdfbeed5>:9	+details	Unknown	Unknown	0/20				
13	reduceByKey at <ipython-input-10-7a8dbdfbeed5>:9	+details	Unknown	Unknown	0/20				
11	reduceByKey at <ipython-input-10-7a8dbdfbeed5>:9	+details	Unknown	Unknown	0/20				

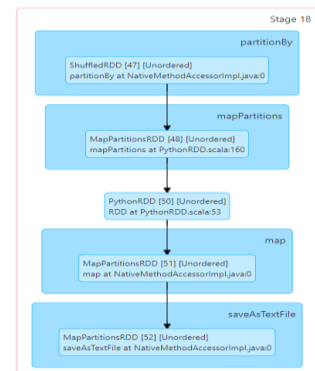Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Here are the 13 completed jobs

Here are DAG Visualization for 15 completed stages

21

Stage 2

textFile
/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/The_Green_hat_pg71913.txt [1]
textFile at NativeMethodAccessorImpl.java:0

/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/The_Green_hat_pg71913.txt [2]
textFile at NativeMethodAccessorImpl.java:0

textFile
/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/The_Australian_aboriginal_pg71940.txt [3]
textFile at NativeMethodAccessorImpl.java:0

/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/The_Australian_aboriginal_pg71940.txt [4]
textFile at NativeMethodAccessorImpl.java:0

textFile
/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/elements_of_meta_physics_pg71885.txt [7]
textFile at NativeMethodAccessorImpl.java:0

/content/drive/MyDrive/Mintesnotf-ASSpark-assignment/books/elements_of_meta_physics_pg71885.txt [8]
textFile at NativeMethodAccessorImpl.java:0

union
UnionRDD [6]
union at NativeMethodAccessorImpl.java:0

union
UnionRDD [10]
union at NativeMethodAccessorImpl.java:0

PythonRDD [13]
RDD at PythonRDD.scala:53

Stage 3

Stage 4

Stage 12

partitionBy
ShuffledRDD [41] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions
MapPartitionsRDD [42] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [44] [Unordered]
sortBy at <ipython-input-10-7a8dbdfbeed5>:6

Stage 14

partitionBy
ShuffledRDD [41] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions
MapPartitionsRDD [42] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [45] [Unordered]
sortBy at <ipython-input-10-7a8dbdfbeed5>:6

PairwiseRDD [46] [Unordered]
sortBy at <ipython-input-10-7a8dbdfbeed5>:6

Stage 15

partitionBy
ShuffledRDD [47] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions
MapPartitionsRDD [48] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [49] [Unordered]
RDD at PythonRDD.scala:53

Stage 18

partitionBy
ShuffledRDD [47] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0

mapPartitions
MapPartitionsRDD [48] [Unordered]
mapPartitions at PythonRDD.scala:160

PythonRDD [50] [Unordered]
RDD at PythonRDD.scala:53

map
MapPartitionsRDD [51] [Unordered]
map at NativeMethodAccessorImpl.java:0

saveAsTextFile
MapPartitionsRDD [52] [Unordered]
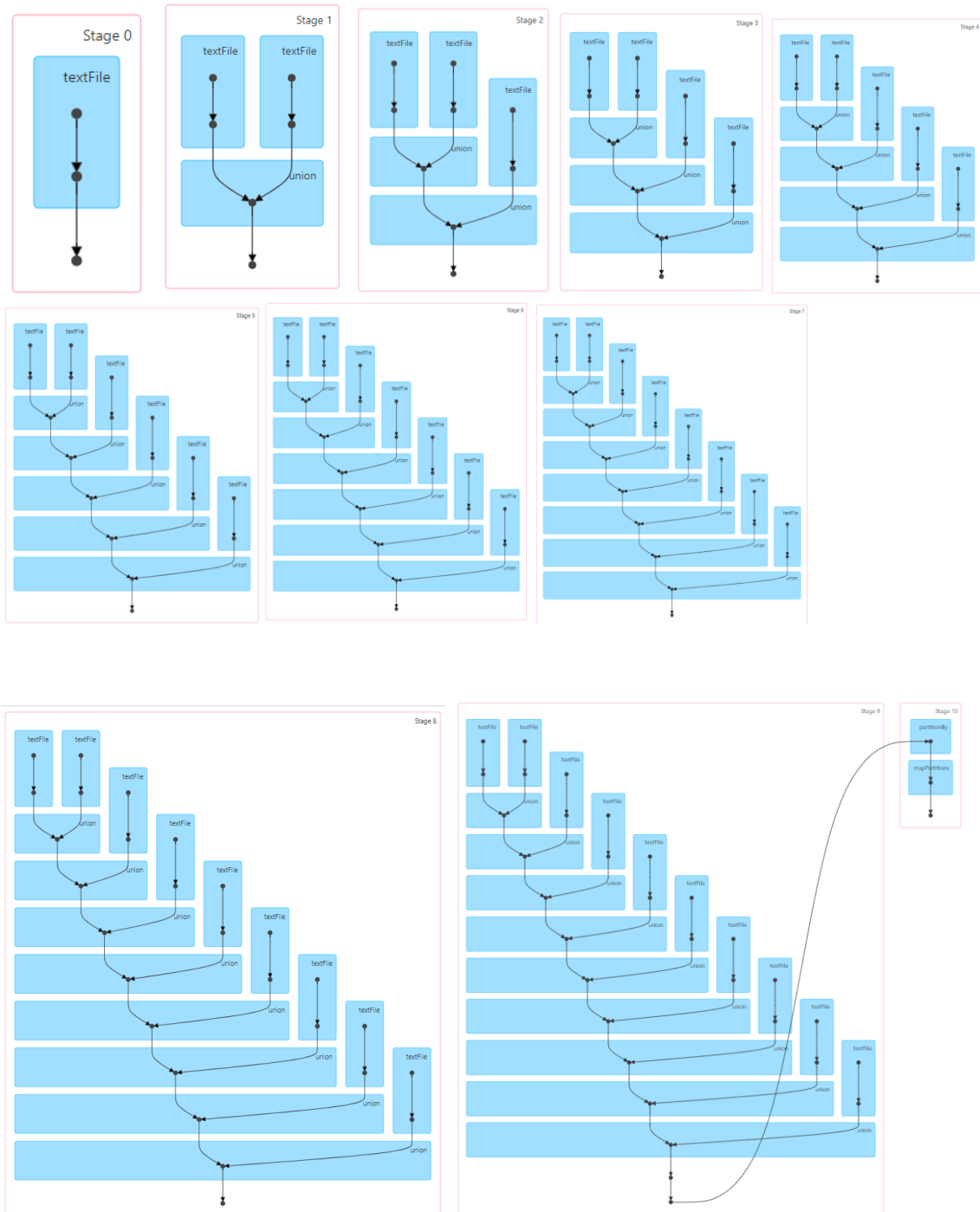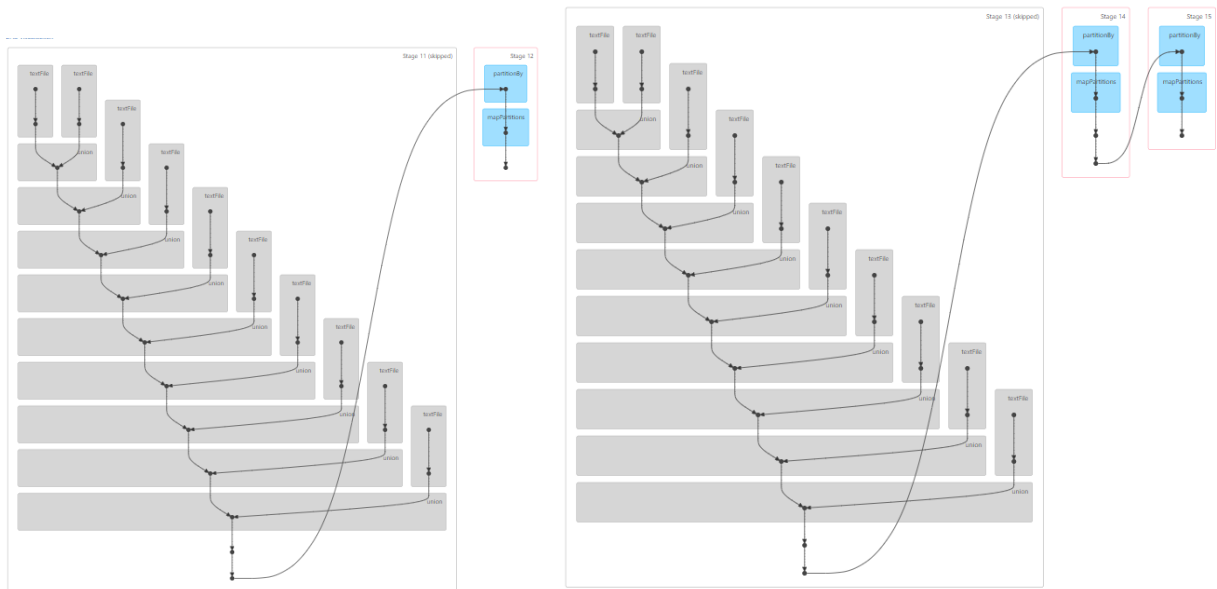saveAsTextFile at NativeMethodAccessorImpl.java:0

Here are DAG Visualization for 13 of them completed jobes sequentially(0 to 12)

Details for Job 12

Status: SUCCEEDED
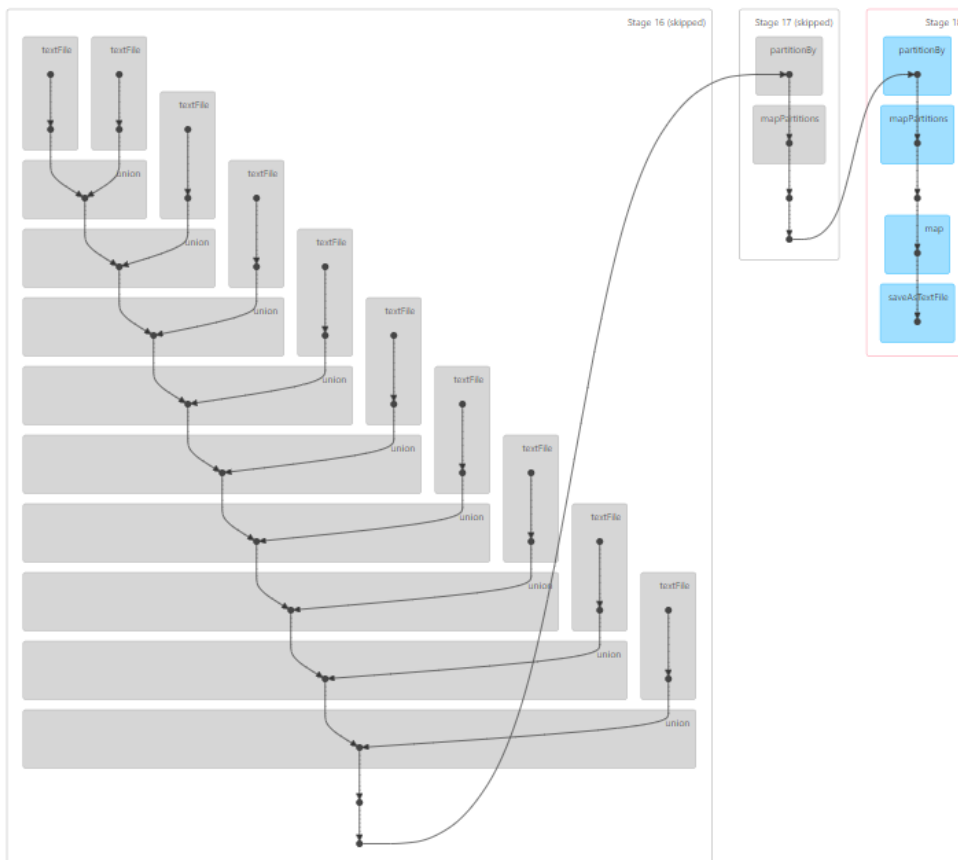**Submitted:** 2024/02/15 11:54:40
Duration: 4 s
Completed Stages: 1
Skipped Stages: 2

▸ Event Timeline
▾ DAG Visualization

The Execution Is Broken Down into 15 Stages, here is the analysis The Above DAG Visualization Result from word occurrence count from 10 books.

In analyzing the execution of the word count application on multiple books from Project Gutenberg using PySpark, the process was divided into several distinct stages as observed through the Spark UI's DAG visualization. Specifically, the execution broke into 19 stages, but 4 of these stages were skipped, leaving only completed stages as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, and 18. The skipped stages indicate operations that were planned but not necessary for the execution, possibly due to optimizations or the absence of required data processing.

The reason for this segmented execution lies in how Spark organizes tasks and data flows. Each stage represents a set of operations that can be performed in parallel, with stages primarily being separated by actions that induce shuffling of data across the cluster. For instance, operations like **textFile** for reading data and **union** for combining RDDs can be grouped within the same stage as they do not require data to be shuffled. However, operations like **reduceByKey** and **sortBy**, which necessitate data movement to ensure that all values for a given key are located on the same partition, trigger the creation of new stages.

The stages that were completed correspond to the following operations in our application:

- Initial reading of the text files and union operations to aggregate data from multiple books into a single RDD (Stages 0-7).

- The transformation steps, including filtering by stopwords and counting occurrences of each unique word (Stages 8-9).

- Sorting operations to organize the word counts in descending order, which involve shuffling to ensure that the sort order is globally maintained (Stages 10, 12, 14, and 15).

- Finally, the **saveAsTextFile** operation, which outputs the results to the file system, represented by Stage 18.

This structured approach, with stages delineated by shuffling boundaries, ensures efficient data processing by minimizing data movement across the cluster and leveraging parallel processing capabilities. The screenshots from the Spark UI's DAG visualization, which I've included, corroborate this analysis by visually depicting the flow and dependencies of these stages, showcasing the complexity and efficiency of distributed data processing with Spark.

Part 1 Question 3

3. Your WordCount application should compute the same results as your WordCount application from Homework #1. Answer the following based on your knowledge of both MapReduce and Spark.

    A. If you were running your WordCount programs in a large cluster or cloud environment, and one of the nodes you were running on died mid computation, how would your MapReduce and Spark programs handle this?

    B. Explain one concrete benefit you experienced when writing the Spark version of WordCount compared to the MapReduce version.

Solution:
Here is the result both for the hadoop and spark implementation of word occurrence count application. Left is result spark from spark implementation and there right is from hadoop

	Word	Count Without Stopwords from all 10 books
1	government	2723
2	united	2559
3	time	2488
4	great	2161
5	volume	1995
6	people	1610
7	general	1596
8	men	1579
9	man	1576
10	good	1395
11	work	1274
12	life	1175
13	thought	1162
14	years	1148
15	south	1132
16	war	1129
17	long	1099
18	british	1078
19	order	1054
20	day	1036
21	american	980
22	number	977
23	hand	955
24	president	950
25	country	942

```
mifaroot@Mifaroot:~/A4mapreduce$ java SortWord local_output/word_count
The top 25 words with highest counts
government - 2723
united - 2559
time - 2488
great - 2161
volume - 1995
people - 1610
general - 1596
men - 1579
man - 1576
good - 1395
work - 1274
life - 1175
thought - 1162
years - 1148
south - 1132
war - 1129
long - 1099
british - 1078
order - 1054
day - 1036
american - 980
number - 977
hand - 955
president - 950
country - 942
mifaroot@Mifaroot:~/A4mapreduce$
```

As we see from the above result on the same dataset both implementation result the same outputs.

    A. *If you were running your WordCount programs in a large cluster or cloud environment, and one of the nodes you were running on died mid computation, how would your MapReduce and Spark programs handle this?*

Answer: When running WordCount programs in a large cluster or cloud environment, both MapReduce and Spark have robust mechanisms for handling node failures, ensuring the resilience and reliability of data processing tasks:

- **MapReduce**: In the event of a node failure, MapReduce automatically redistributes the tasks of the failed node to other nodes in the cluster. Thanks to HDFS's replication mechanism, data processed or stored by the failed node is available from other nodes, allowing the job to continue without data loss. This redundancy ensures that MapReduce jobs can recover from node failures,

albeit with potential increases in processing time due to the redistribution of tasks and data retrieval from other nodes.

- **Spark**: Spark handles node failure through its RDD lineage concept. If a node fails during computation, Spark can recompute the lost partitions of an RDD on other nodes, using the lineage information (the series of transformations that led to the RDD). This ability to recompute data on-the-fly, combined with data caching in memory, often results in quicker recovery times compared to the disk-based recovery mechanisms of MapReduce. However, for large datasets that don't fit entirely in memory, Spark might also experience performance hits during recovery, similar to MapReduce.

B. Explain one concrete benefit you experienced when writing the Spark version of WordCount compared to the MapReduce version.

Answer:

One significant benefit I experienced when implementing the WordCount application in Spark compared to MapReduce was the speed and efficiency of development and execution. Spark's in-memory processing capabilities vastly outpaced the disk-based operations of MapReduce, leading to faster data processing times for the WordCount task. This was particularly evident in iterative operations where Spark could quickly access cached data stored in memory, unlike MapReduce, which had to read data from and write data to the disk at each step.

Additionally, configuring Spark and running it on my personal computer (even when installed on an Ubuntu OS) was more straightforward and less resource-intensive compared to setting up Hadoop MapReduce, which was challenging and slowed down my system significantly. The difference in system performance was notable; Spark's efficient use of RAM for processing made it not only faster but also safer for my PC's SSD, in contrast to the disk-intensive operations of Hadoop that contributed to system slowdowns. This ease of setup and reduced impact on system performance greatly enhanced my productivity and overall experience with the Spark version of the WordCount application.

Part 2 - Code Analysis

Read through the following PySpark code snippet and answer the questions following:

```
1    lines = sc.textFile(file)
2    links = lines.map(lambda urls: parseNeighbors(urls)) \
3        .groupByKey() \
4        .cache()
5    N = links.count()
6    ranks = links.map(lambda u: (u[0], 1.0/N))
7
8    for i in range(iters):
9        contribs = links.join(ranks) \
10           .flatMap(lambda u: computeContribs(u[1][0], u[1][1]))
11
12       ranks = contribs.reduceByKey(lambda a,b: a+b) \
13           .mapValues(lambda rank: rank * 0.85 + 0.15*(1.0/N))
14   return ranks
```

4. Given the above spark application, draw the lineage graph DAG for the RDD ranks on line 12 when the iteration variable i has a value of 2. Include nodes for all intermediate RDDs, even if they are unnamed.

5. How many stages will the above DAG be broken into? Give the number of stages AND draw stage boundaries on your diagram.

6. Identify in the above code (by function name AND line number) one instance of:
   a) A transformation that results in a wide dependency
   b) A transformation that results in a narrow dependency
   c) A transformation that may result in a narrow dependency OR a wide dependency
   d) An action

7. How many "jobs" will the above code run if iters has value 10?

8. What algorithm is the above code an implementation of?

Solution:
4. Given the above spark application, draw the lineage graph DAG for the RDD ranks on line 12 when the iteration variable i has a value of 2. Include nodes for all intermediate RDDs, even if they are unnamed.
Answer:
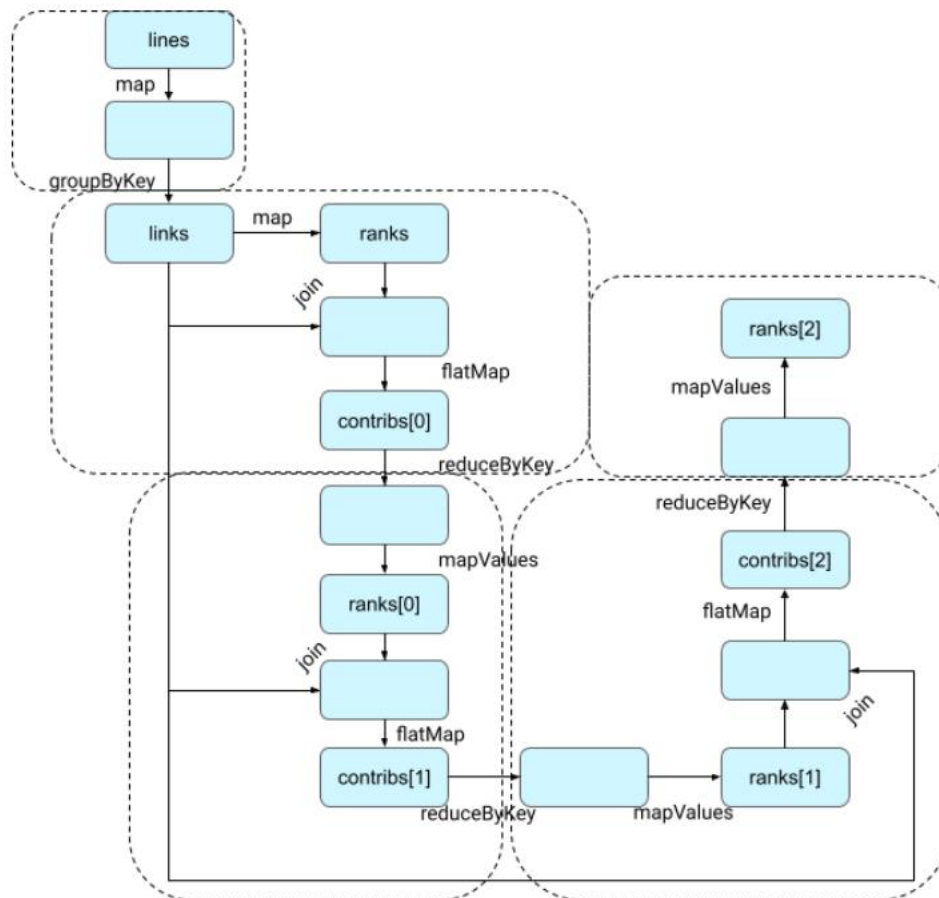
5. How many stages will the above DAG be broken into? Give the number of stages AND draw stage boundaries on your diagram.

Answer: 8 stages
The above DAG will be broken into multiple stages, primarily due to the presence of wide transformations (**groupByKey**, **reduceByKey**, and potentially **join**). For each iteration, the **join** and **reduceByKey** operations mark new stage boundaries because of the shuffling they necessitate.

- Initial reading and processing of **lines** into **links** forms the first stage, ending at **groupByKey**.
- The calculation of initial **ranks** may not necessarily introduce a new stage since it follows narrow transformations (**map**).
- Each iteration introduces at least two stages:
  - One for the **join** operation (if considered wide by Spark's execution plan).
  - One for the **reduceByKey** transformation.

Given two complete iterations (**i=2**), and considering **join** as a wide transformation, we could expect up to 5 stages for the first iteration (including the initial read and processing stage) and an additional 3 stages for the second iteration, totaling up to 8 stages.

However, if **join** operations are optimized by Spark and not considered as introducing new stages, the total might be closer to 5 stages, with the initial stage followed by two stages per iteration focused around the **reduceByKey** transformations.

Therefore, the exact number of stages can vary based on Spark's optimization of **join** operations but expect a minimum of 5 stages, with the potential for up to 8 if each **join** is treated as a wide transformation.

6. Identify in the above code (by function name AND line number) one instance of:
   a. A transformation that results in a wide dependency
   b. A transformation that results in a narrow dependency
   c. A transformation that may result in a narrow dependency OR a wide dependency
   d. An action

Answer:
a. Wide Dependency Transformation:
   - **Transformation: reduceByKey** (Line 12).
   - **Description:** This operation aggregates values for each key using a specified reduce function, which results in shuffling data across the cluster. Shuffling is necessary to bring together values across partitions that share the same key, thus creating a "wide" dependency where multiple parts of the data are consolidated.
b. Narrow Dependency Transformation:
   - **Transformation: mapValues** (Line 13).
   - **Description:** Applies a function to each value of a pair RDD while maintaining the same keys. This operation does not require shuffling data across partitions, as it operates independently on each partition. The dependency remains "narrow" since it does not involve data movement across partitions.
c. Variable Dependency Transformation:
   - **Transformation: join** (Line 9).
   - **Description:** Merges two RDDs by key. The nature of the dependency (wide or narrow) can vary based on the distribution and partitioning of the dataset. If the operation requires significant data movement to co-locate matching keys, it results in a wide dependency. However, if keys are already partitioned in a manner that minimizes data movement, the dependency could be considered narrow.
d. Action:
   - **Action: count** (Line 5).
   - **Description:** Triggers the execution of the RDD lineage graph to return the number of elements in the RDD. As an action, **count** causes Spark to evaluate transformations that have been lazily defined up to this point, initiating job execution and computation of results.

7. How many "jobs" will the above code run if iters has value 10?

Answer: The code will run 1 job. Despite the iterative process defined within the loop, Spark's lazy evaluation model means that transformations are not executed until an action is called. Since there's only one action (**count** on line 5) outside the iterative loop that triggers the evaluation of the RDD transformations, only one job is initiated regardless of the number of iterations specified by **iters**. It's important to note that this interpretation assumes that the final ranks RDD is acted upon outside the provided code snippet, as actions like **collect ()** or **saveAsTextFile ()** on **ranks** after the loop would indeed trigger additional jobs corresponding to each iteration's transformations.

8. What algorithm is the above code an implementation of?

Answer: The provided code is an implementation of the **PageRank algorithm**. PageRank is a link analysis algorithm used to determine the importance of website pages by measuring the number and

quality of links to a page. This algorithm essentially assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of measuring its relative importance within the set. The iterative process in the code, which adjusts ranks based on the contributions from linked nodes and normalizes these ranks, aligns with the core principles of the **PageRank** computation