Basic Word Count

In the first phase of the project, we built a basic word count program using **Apache Spark**. The program reads a text file, counts the frequency of each word, and outputs the results as a list of key-value pairs, where the key is the word and the value is the number of times it appears in the file.

Dataset source:
Project Gutenberg (https://www.gutenberg.org/ebooks/)

Extended Word Count:
In addition to basic word counting, we extended our code, to do the following:

1. It must be case-insensitive (lower() in Python)
2. It must ignore all punctuation (translate() in Python)
3. It must ignore stopwords (filter() in Spark)
4. The output must be sorted by count in descending order (sortBy() in Spark)

To accomplish the above tasks, we took reference code from the below-mentioned link.
RDD Programming Guide - Spark 4.0.1 Documentation

Analysis:

Analysis 1: Basic Word Count

The first run of the program was executed in the PySpark REPL on a **single text file** to analyze the **raw frequency distribution of words**.

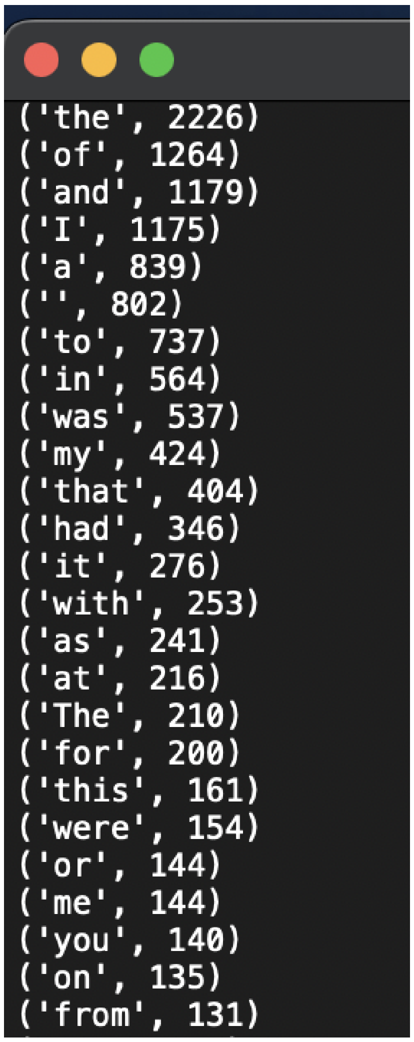
- **What we are analyzing**: The top 25 most frequent words in the text, without applying any preprocessing.

- **Why it matters**: This helps us understand the natural distribution of words before applying transformations like case normalization, punctuation removal, or stopword filtering. It also provides a baseline for comparison with the extended word count results.

- **Key observation**: As expected, common stopwords such as *the, of, and, I, a, to* dominate the list. The presence of both lowercase and capitalized words (*the* vs *The*) and even empty strings (`''`) highlights the need for preprocessing steps in large-scale text analysis.

Top 25 words (raw count):

```
[
        ('the', 2226)
        ('of', 1264)
        ('and', 1179)
        ('I', 1175)
        ('a', 839)
        ('', 802)
        ('to', 737)
        ('in', 564)
        ('was', 537)
        ('my', 424)
        ('that', 404)
        ('had', 346)
        ('it', 276)
        ('with', 253)
        ('as', 241)
        ('at', 216)
        ('The', 210)
        ('for', 200)
        ('this', 161)
        ('were', 154)
        ('or', 144)
        ('me', 144)
        ('you', 140)
        ('on', 135)
        ('from', 131)
]
```



```
('the', 2226)
('of', 1264)
('and', 1179)
('I', 1175)
('a', 839)
('', 802)
('to', 737)
('in', 564)
('was', 537)
('my', 424)
('that', 404)
('had', 346)
('it', 276)
('with', 253)
('as', 241)
('at', 216)
('The', 210)
('for', 200)
('this', 161)
('were', 154)
('or', 144)
('me', 144)
('you', 140)
('on', 135)
('from', 131)
```

Analysis 2: Execution Stages in Spark Word Count

The PySpark word count program was executed on a single text file, and the execution plan was analyzed through Spark's WebUI DAG visualization.

What we are analyzing:
The number of stages Spark breaks the job into and the reasoning behind this stage division.

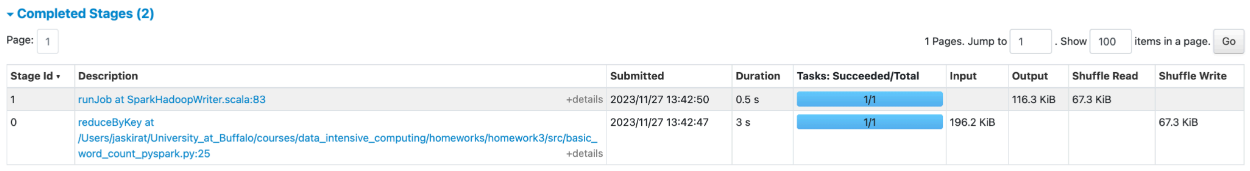
Why it matters:
Understanding how Spark divides work into stages helps in optimizing performance and managing data shuffling efficiently. Each stage consists of tasks that can run in parallel without needing data from other stages. Recognizing the impact of wide dependencies (e.g., `reduceByKey`) allows for better anticipation of network shuffles and potential bottlenecks.

Key observations:

1. **Stages identified:** The execution was broken into **1 job and 2 stages**:
   - **Stage 1:** Reading the text file and performing the `reduceByKey` operation.
   - **Stage 2:** Executing `partitionBy`, `mapPartitions`, `map`, `coalesce`, and `saveAsTextFile`.
2. **Reason for stage division:**
   - Spark uses **lazy evaluation**, meaning it builds a DAG of transformations only when an action is triggered.
   - **Wide dependencies**, such as those introduced by `reduceByKey`, require data to be shuffled across partitions, creating new stages.
   - Operations like `partitionBy`, `mapPartitions`, `map`, `coalesce`, and `saveAsTextFile` do not require shuffles, so they are grouped into a single stage.
3. **Impact of data size:**
   - In this run, the dataset is small enough that a single stage can handle multiple transformations sequentially without additional shuffles.

<u>Conclusion:</u>
Spark divides execution into stages based on dependencies between RDDs. Narrow transformations can be executed together, while wide transformations that require data shuffling define the boundaries between stages. Understanding this helps optimize distributed computation and minimize unnecessary network overhead.

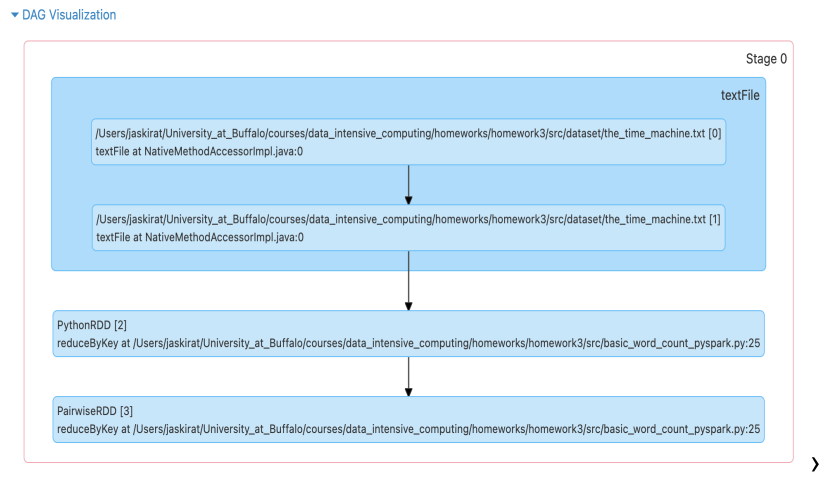
<u>Supporting evidence:</u>
The screenshots of the DAG from Spark's WebUI would illustrate the two stages and the task flow.

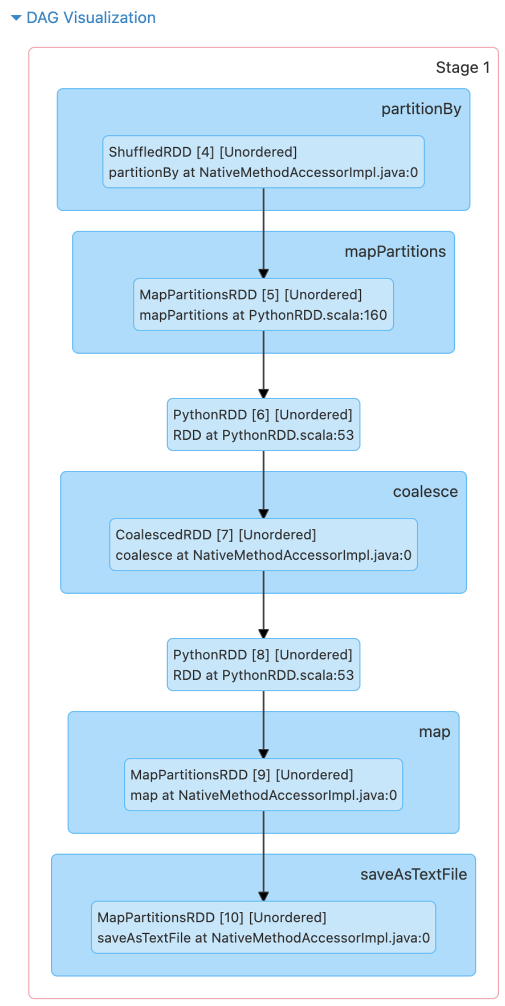


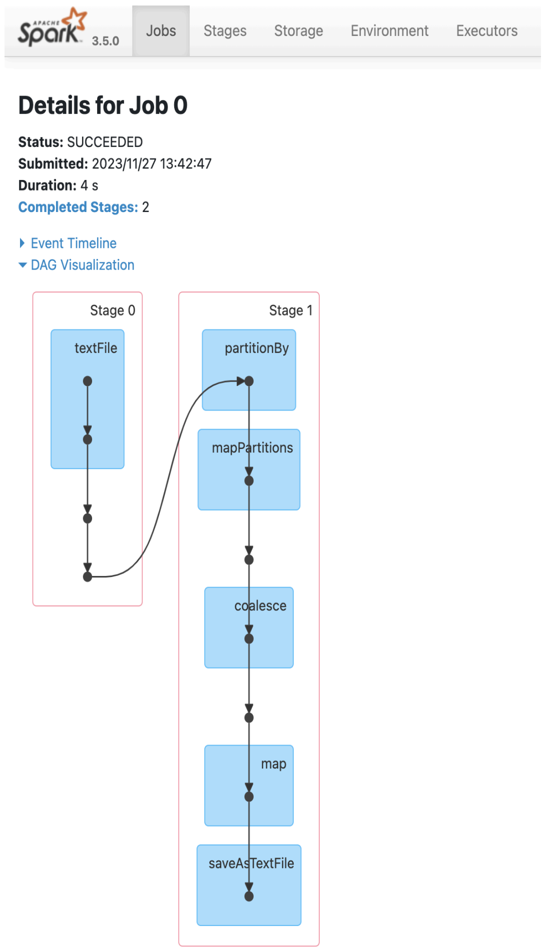
Completed Stages (2)

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
1	runJob at SparkHadoopWriter.scala:83 +details	2023/11/27 13:42:50	0.5 s	1/1		116.3 KiB	67.3 KiB	
0	reduceByKey at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/basic_word_count_pyspark.py:25 +details	2023/11/27 13:42:47	3 s	1/1	196.2 KiB			67.3 KiB


DAG Visualization
Stage 0
textFile
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/the_time_machine.txt [0]
textFile at NativeMethodAccessorImpl.java:0
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/the_time_machine.txt [1]
textFile at NativeMethodAccessorImpl.java:0
PythonRDD [2]
reduceByKey at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/basic_word_count_pyspark.py:25
PairwiseRDD [3]
reduceByKey at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/basic_word_count_pyspark.py:25


DAG Visualization
Stage 1
partitionBy
ShuffledRDD [4] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [5] [Unordered]
mapPartitions at PythonRDD.scala:160
PythonRDD [6] [Unordered]
RDD at PythonRDD.scala:53
coalesce
CoalescedRDD [7] [Unordered]
coalesce at NativeMethodAccessorImpl.java:0
PythonRDD [8] [Unordered]
RDD at PythonRDD.scala:53
map
MapPartitionsRDD [9] [Unordered]
map at NativeMethodAccessorImpl.java:0
saveAsTextFile
MapPartitionsRDD [10] [Unordered]
saveAsTextFile at NativeMethodAccessorImpl.java:0



Details for Job 0

Status: SUCCEEDED
Submitted: 2023/11/27 13:42:47
Duration: 4 s
Completed Stages: 2

Analysis 3: Extended Word Count Across Two Text Files

The extended PySpark word count program was executed on **two text files** to analyze word frequency across a larger corpus.

What we are analyzing:

The top 25 most frequent words after applying preprocessing steps such as:

- Removing punctuation
- Normalizing case (making text case-insensitive)
- Removing stopwords

- Sorting results by descending frequency

Why it matters:
By combining two text files into a single RDD, we gain insights into word distribution across multiple sources rather than a single file. Preprocessing ensures that word counts are meaningful, avoiding inflated counts due to capitalization, punctuation, or common filler words.

Key observations:

1. **Data preparation:** The two books were read separately and then merged using a union operation to form a single RDD.

2. **Transformation impact:** Removing stopwords and normalizing case reduced noise, highlighting the actual significant words across the corpus.

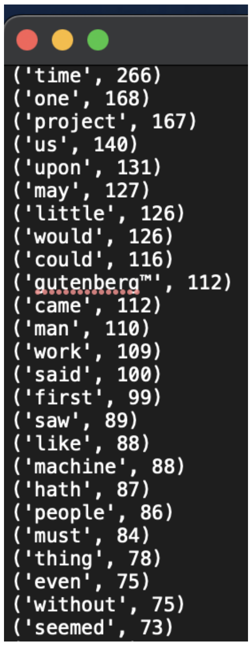
3. **Result:** The top 25 words reflect meaningful content rather than common filler words, giving a clearer picture of key terms used in both texts.

Supporting evidence:
A screenshot of the program output demonstrates the top 25 words and their counts.

Additional notes:
The code used for this analysis is available in the `src` folder as `extended_word_count_pyspark.py`, and the input text files are located in the `dataset` folder.

25 most common words:

```
[
        ('time', 266)
        ('one', 168)
        ('project', 167)
        ('us', 140)
        ('upon', 131)
        ('may', 127)
        ('little', 126)
        ('would', 126)
        ('could', 116)
```



```
('time', 266)
('one', 168)
('project', 167)
('us', 140)
('upon', 131)
('may', 127)
('little', 126)
('would', 126)
('could', 116)
('gutenberg™', 112)
('came', 112)
('man', 110)
('work', 109)
('said', 100)
('first', 99)
('saw', 89)
('like', 88)
('machine', 88)
('hath', 87)
('people', 86)
('must', 84)
('thing', 78)
('even', 75)
('without', 75)
('seemed', 73)
```

```
        ('gutenberg™', 112)
        ('came', 112)
        ('man', 110)
        ('work', 109)
        ('said', 100)
        ('first', 99)
        ('saw', 89)
        ('like', 88)
        ('machine', 88)
        ('hath', 87)
        ('people', 86)
        ('must', 84)
        ('thing', 78)
        ('even', 75)
        ('without', 75)
        ('seemed', 73)
    ]
```

Analysis 4: Execution Stages in Extended Word Count

The extended PySpark word count program was executed on **two text files**, and the execution plan was analyzed using Spark's WebUI DAG visualization.

What we are analyzing:

The number of stages Spark divides the job into and the reasoning behind this stage segmentation when processing a larger dataset.

Why it matters:

Understanding Spark's stage division helps optimize distributed computation. Each stage consists of tasks that can execute in parallel without requiring data from other stages. Wide dependencies (like `reduceByKey` and `sortBy`) trigger shuffles across partitions, creating new stages.

Key observations:

1. **Stages identified:** The execution was broken into **3 jobs and 6 stages**:

   - **Stage 0:** Reading the two text files, performing a `union`, and executing `reduceByKey`.

    - **Stages 1–4:** Executing sequences of `partitionBy`, `mapPartitions`, and `sortBy` operations.
    - **Stage 5:** Final transformations including `partitionBy`, `mapPartitions`, `coalesce`, `map`, and `saveAsTextFile`.

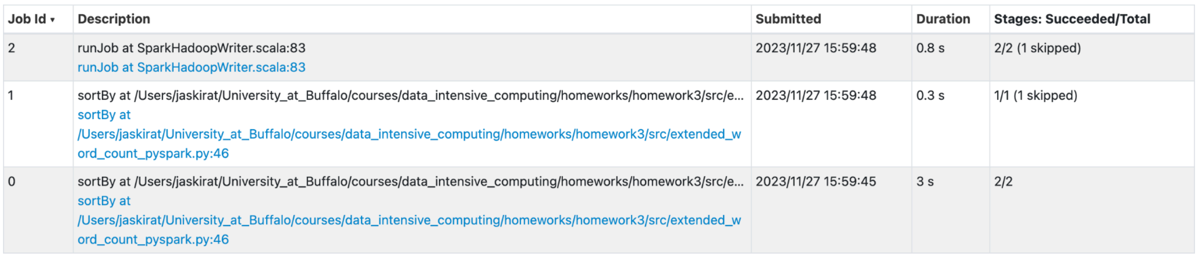
2. **Reason for stage division:**
    - Spark uses **lazy evaluation**: a DAG of transformations is only executed when an action is triggered.
    - **Wide dependencies**, such as `reduceByKey` and `sortBy`, require shuffling data across partitions, resulting in new stages.
    - Compared to processing a single file, the **larger combined dataset** introduces more parallelism, which distributes transformations across multiple stages.

3. **Impact of data size and transformations:**
    - Operations that can run without shuffles are grouped into one stage.
    - Union of two RDDs and subsequent sorting requires splitting into multiple stages to efficiently manage parallel execution.
    - The final stage consolidates results and writes the output using `saveAsTextFile`.

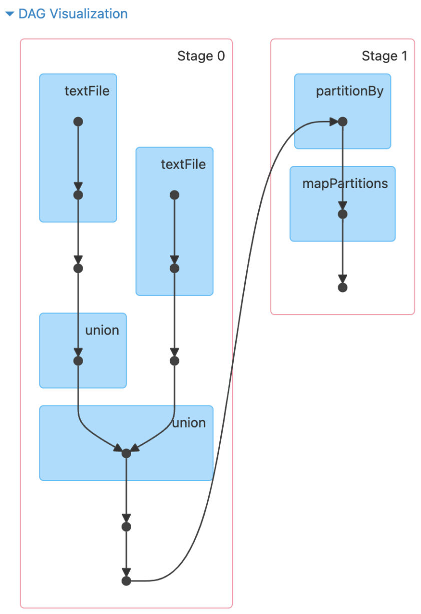
Conclusion:
Spark divides execution into stages based on dependencies between transformations and data shuffling requirements. Larger datasets increase the number of stages to leverage parallel computation effectively.

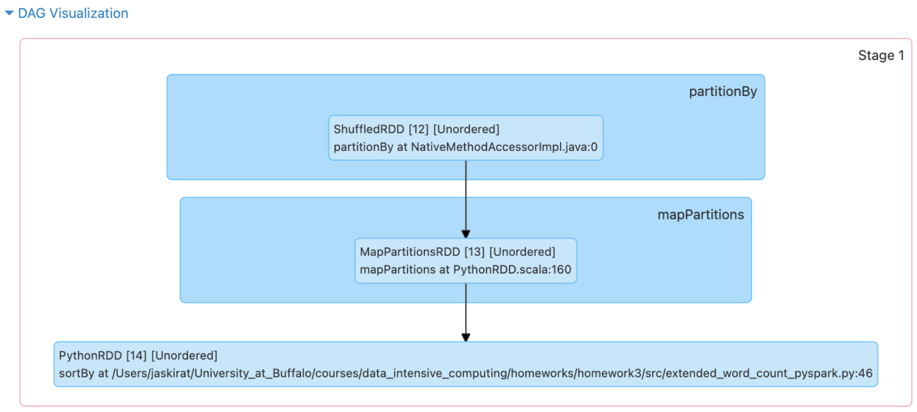
Supporting evidence:
A screenshot of the DAG from Spark's WebUI would illustrate the six stages and task flow.

Screenshots -



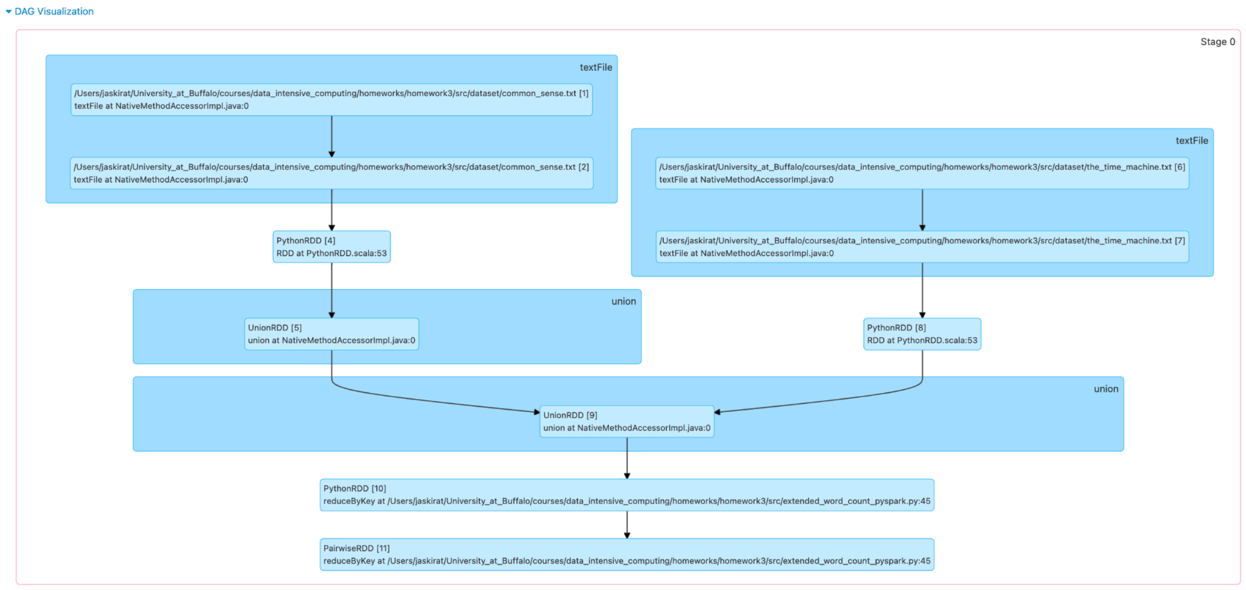
Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total
2	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2023/11/27 15:59:48	0.8 s	2/2 (1 skipped)
1	sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/e... sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:46	2023/11/27 15:59:48	0.3 s	1/1 (1 skipped)
0	sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/e... sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:46	2023/11/27 15:59:45	3 s	2/2

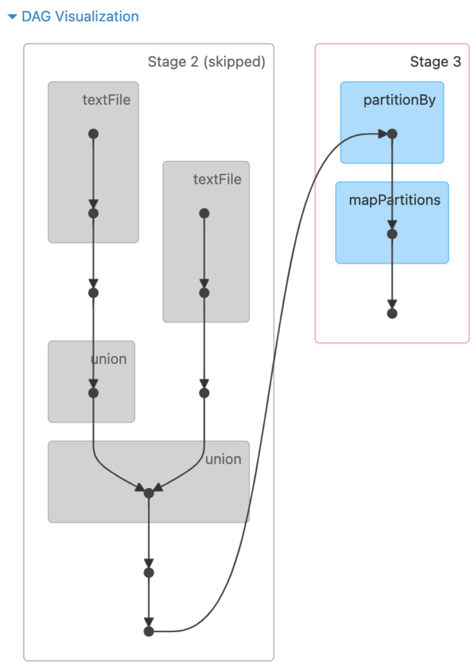


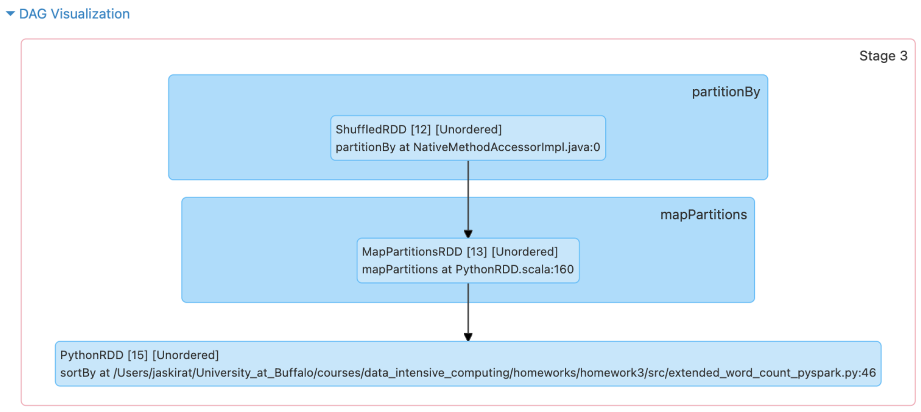

DAG Visualization
Stage 1
partitionBy
ShuffledRDD [12] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [13] [Unordered]
mapPartitions at PythonRDD.scala:160
PythonRDD [14] [Unordered]
sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:46

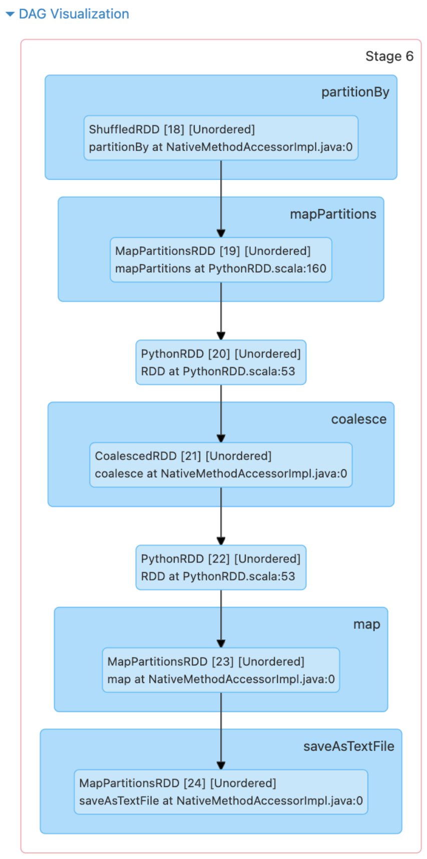
Completed Stages (2)

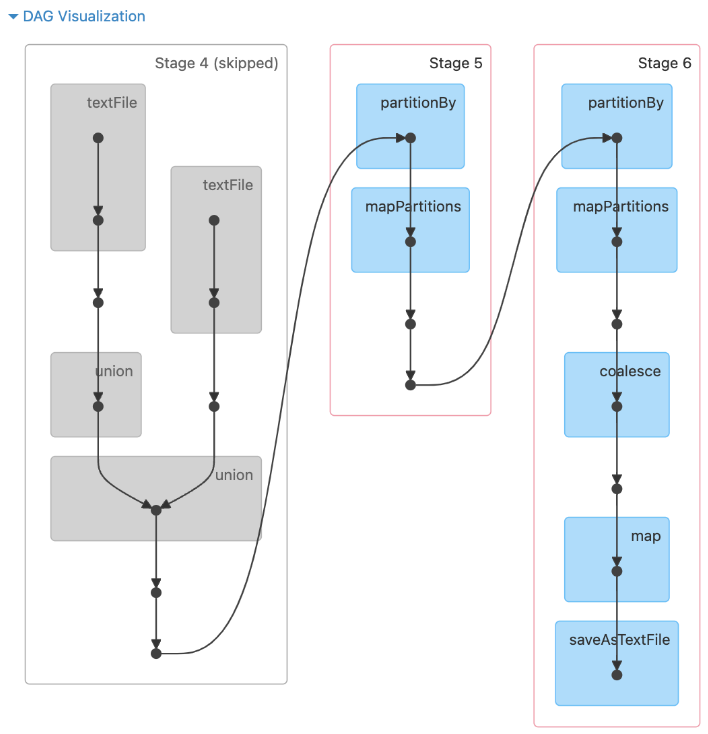
Page: 1


DAG Visualization
Stage 0
textFile
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/common_sense.txt [1]
textFile at NativeMethodAccessorImpl.java:0
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/common_sense.txt [2]
textFile at NativeMethodAccessorImpl.java:0
textFile
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/the_time_machine.txt [6]
textFile at NativeMethodAccessorImpl.java:0
/Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/dataset/the_time_machine.txt [7]
textFile at NativeMethodAccessorImpl.java:0
PythonRDD [4]
RDD at PythonRDD.scala:53
union
UnionRDD [5]
union at NativeMethodAccessorImpl.java:0
PythonRDD [8]
RDD at PythonRDD.scala:53
union
UnionRDD [9]
union at NativeMethodAccessorImpl.java:0
PythonRDD [10]
reduceByKey at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:45
PairwiseRDD [11]
reduceByKey at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:45


DAG Visualization
Stage 2 (skipped)
textFile
textFile
union
union
Stage 3
partitionBy
mapPartitions


DAG Visualization
Stage 3
partitionBy
ShuffledRDD [12] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [13] [Unordered]
mapPartitions at PythonRDD.scala:160
PythonRDD [15] [Unordered]
sortBy at /Users/jaskirat/University_at_Buffalo/courses/data_intensive_computing/homeworks/homework3/src/extended_word_count_pyspark.py:46


DAG Visualization
Stage 6
partitionBy
ShuffledRDD [18] [Unordered]
partitionBy at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [19] [Unordered]
mapPartitions at PythonRDD.scala:160
PythonRDD [20] [Unordered]
RDD at PythonRDD.scala:53
coalesce
CoalescedRDD [21] [Unordered]
coalesce at NativeMethodAccessorImpl.java:0
PythonRDD [22] [Unordered]
RDD at PythonRDD.scala:53
map
MapPartitionsRDD [23] [Unordered]
map at NativeMethodAccessorImpl.java:0
saveAsTextFile
MapPartitionsRDD [24] [Unordered]
saveAsTextFile at NativeMethodAccessorImpl.java:0



Analysis 5: Lineage Graph (DAG) for RDD ranks at Iteration i=2

In the given PySpark application, the lineage graph was analyzed to understand how RDDs are derived during iterative computation, specifically on line 12 when the iteration variable `i = 2`.

What we are analyzing:

The lineage graph (DAG) of the `ranks` RDD, including all intermediate RDDs, to trace data transformations across iterations.

Why it matters:

RDD lineage provides a complete picture of how data flows through transformations, which is essential for:

- Understanding fault tolerance (how Spark can recompute lost partitions)
- Optimizing computation by identifying redundant or expensive transformations

- Visualizing iterative operations, such as those in PageRank or other iterative algorithms

Key observations:

1. RDD derivation at iteration i=2:
    - The `ranks` RDD depends on transformations applied in previous iterations (`i=0` and `i=1`).
    - Intermediate RDDs are created during operations like `join`, `map`, `reduceByKey`, or `aggregate`.
    - Even unnamed RDDs (temporary results) form nodes in the DAG, showing the complete lineage from the original input RDDs to the current `ranks`.
2. DAG structure:
    - The DAG shows a directed acyclic flow, starting from the base RDDs (e.g., adjacency list or input data).
    - Each transformation creates a new node, with edges representing dependencies.
    - The node for `ranks` at i=2 aggregates data from prior iteration nodes, illustrating the iterative computation.

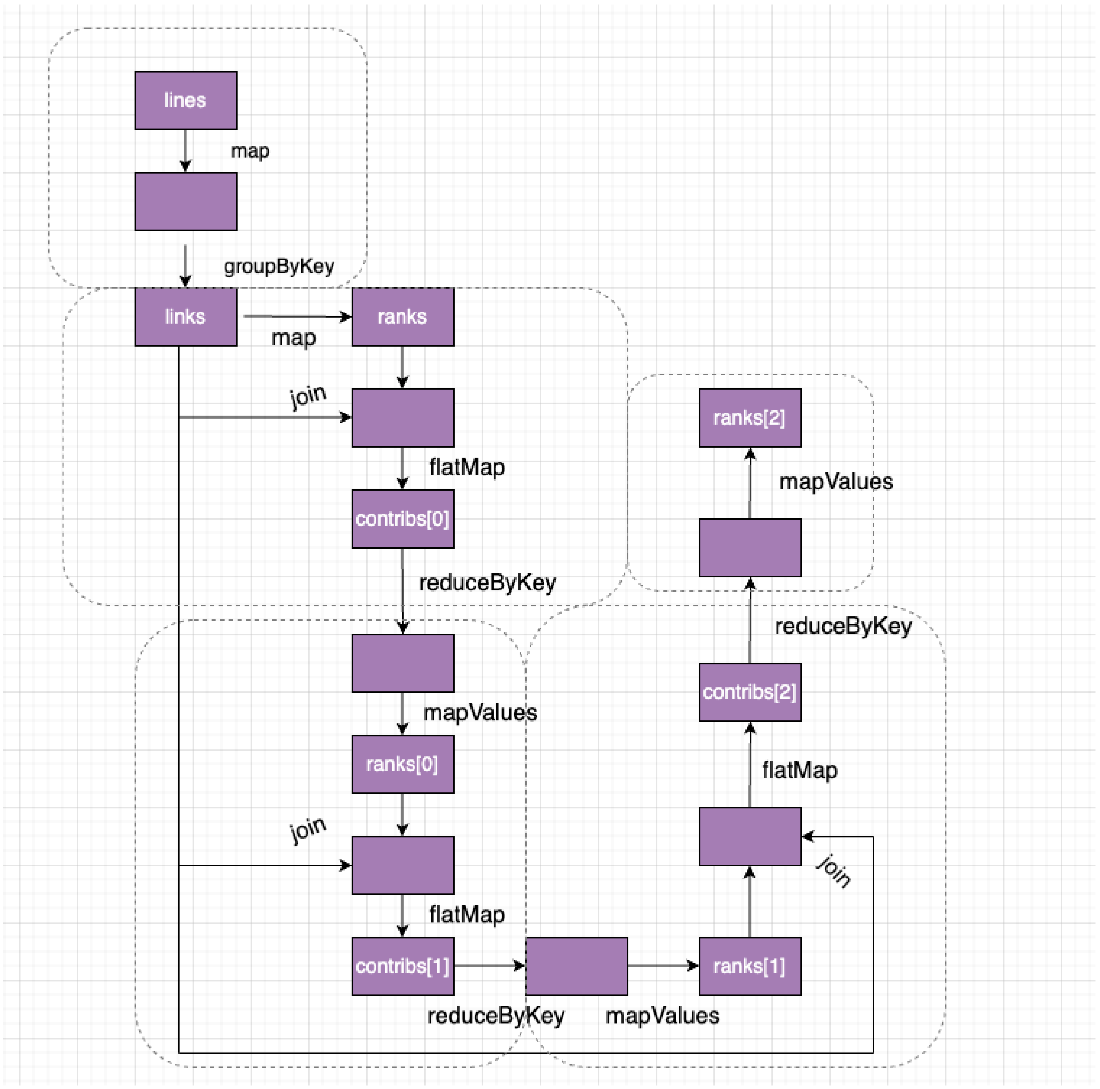
Conclusion:
RDD lineage graphs provide a transparent view of iterative transformations and dependencies in Spark. For iterative algorithms, analyzing DAGs helps in identifying potential optimizations and understanding how Spark maintains fault tolerance by recomputing intermediate RDDs if needed.

Supporting evidence:
A diagram of the lineage graph for `ranks` at iteration i=2 would include all intermediate RDD nodes and show the flow of data through transformations up to the current state.

```python
lines = sc.textFile(file)
links = lines.map(lambda urls: parseNeighbors(urls)) \
             .groupByKey()
             .cache()
N = links.count()
ranks = links.map(lambda u: (u[0], 1.0/N))

for i in range(iters):
  contribs = links.join(ranks) \
       .flatMap(lambda u: computeContribs(u[1][0], u[1][1]))

  ranks = contribs.reduceByKey(lambda a,b: a+b) \
       .mapValues(lambda rank: rank * 0.85 + 0.15*(1.0/N))
return ranks
```


lines
map
groupByKey
links
map
ranks
join
flatMap
contribs[0]
reduceByKey
mapValues
ranks[0]
join
flatMap
contribs[1]
reduceByKey
mapValues
ranks[1]
join
flatMap
contribs[2]
reduceByKey
mapValues
ranks[2]