

Homework #2 – Spark Report

Tharun Teja Mogili - 50559877

Introduction:

This report delves into the practical application of Spark, focusing on the implementation and analysis of fundamental graph and data processing algorithms. Spark, a powerful distributed computing framework, is widely used for handling big data efficiently.

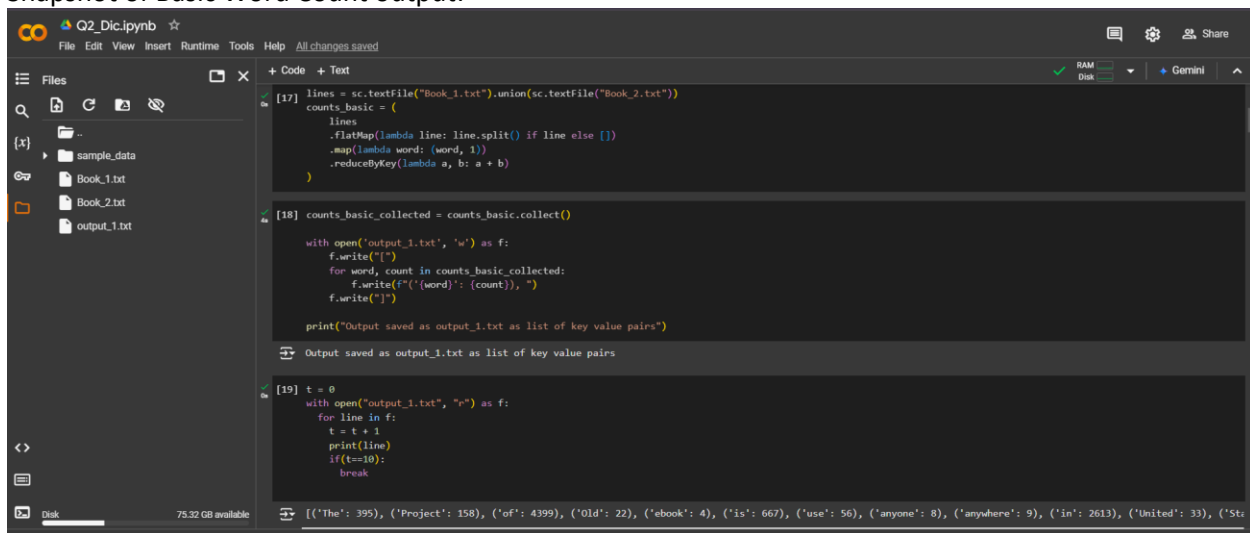
The report is structured around three essential tasks: In the Word Count task, I applied data-cleaning steps like case normalization, punctuation removal, and stop-word filtering using the NLTK library, then sorted the word counts in descending order. For Dijkstra's algorithm, I calculated the shortest path between nodes, identifying the ones with the longest and shortest distances from a starting point. The final task used the PageRank algorithm on a simulated network of webpages, helping me analyze the highest- and lowest-ranked pages based on link structure. Throughout, I used the Ngrok library to visualize data stages in Spark's Directed Acyclic Graph (DAG).

Part -1 Implement and analyze word count.

Basic Word Count Implementation:

The code snippet reads two text files, "Book_1.txt" and "Book_2.txt," using Spark's `textFile()` function and combines them using the `union()` method. Each line of text is split into words, and a key-value pair is created for each word, initializing the count as 1. The `reduceByKey()` function then aggregates these counts to produce the total occurrences for each word. The results are collected and written into "output_1.txt" in a key-value format, where each entry is represented as a tuple ('word': count). This output is saved for further analysis.

Snapshot of Basic Word Count output:



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files: Book_1.txt, Book_2.txt, and output_1.txt. The code editor contains three code cells. The first cell (index 17) reads two text files and combines them. The second cell (index 18) collects the counts and writes them to output_1.txt. The third cell (index 19) reads the output file and prints the first 10 lines. The output of the third cell is visible at the bottom of the notebook.

```
[17] lines = sc.textFile("Book_1.txt").union(sc.textFile("Book_2.txt"))
counts_basic = (
    lines
    .flatMap(lambda line: line.split() if line else [])
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
)

[18] counts_basic_collected = counts_basic.collect()

with open('output_1.txt', 'w') as f:
    f.write("[")
    for word, count in counts_basic_collected:
        f.write(f"('{word}': {count}), ")
    f.write("]")

print("Output saved as output_1.txt as list of key value pairs")

Output saved as output_1.txt as list of key value pairs

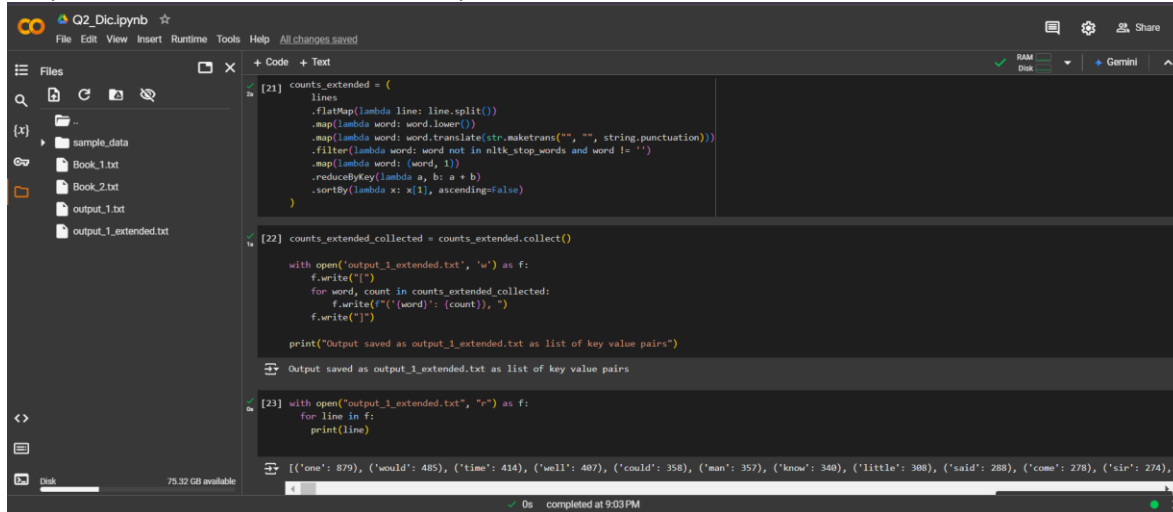
[19] t = 0
with open("output_1.txt", "r") as f:
    for line in f:
        t = t + 1
        print(line)
        if(t==10):
            break
```

[[('The': 395), ('Project': 158), ('of': 4399), ('old': 22), ('ebook': 4), ('is': 667), ('use': 56), ('anyone': 8), ('anywhere': 9), ('in': 2613), ('United': 33), ('St':

Extended Word Count Implementation:

The extended code enhances the basic word count by making it case insensitive, removing punctuation, filtering out stop words using NLTK, and sorting the output by word frequency in descending order. Words are first converted to lowercase to ensure uniformity. Punctuation is stripped using Python's `translate()` method, and common stop words from the NLTK library are excluded using the `filter()` method in PySpark. Finally, the `reduceByKey()` function aggregates the word counts, and `sortBy()` is used to order the words by their count, from highest to lowest. The output is saved in "output_1_extended.txt" as a list of key-value pairs.

Snapshot of extended Word Count Implementation:



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory named 'sample_data' containing files 'Book_1.txt', 'Book_2.txt', 'output_1.txt', and 'output_1_extended.txt'. The code editor contains the following Python code:

```
[21] counts_extended = (
    lines
    .flatMap(lambda line: line.split())
    .map(lambda word: word.lower())
    .map(lambda word: word.translate(str.maketrans("", "", string.punctuation)))
    .filter(lambda word: word not in nltk_stop_words and word != '')
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
    .sortBy(lambda x: x[1], ascending=False)
)

[22] counts_extended_collected = counts_extended.collect()

with open('output_1_extended.txt', 'w') as f:
    f.write("\n")
    for word, count in counts_extended_collected:
        f.write(f"{word}: {count}, ")
    f.write("\n")

print("Output saved as output_1_extended.txt as list of key value pairs")

[23] with open("output_1_extended.txt", "r") as f:
    for line in f:
        print(line)
```

The output of the code is displayed below the code cells:

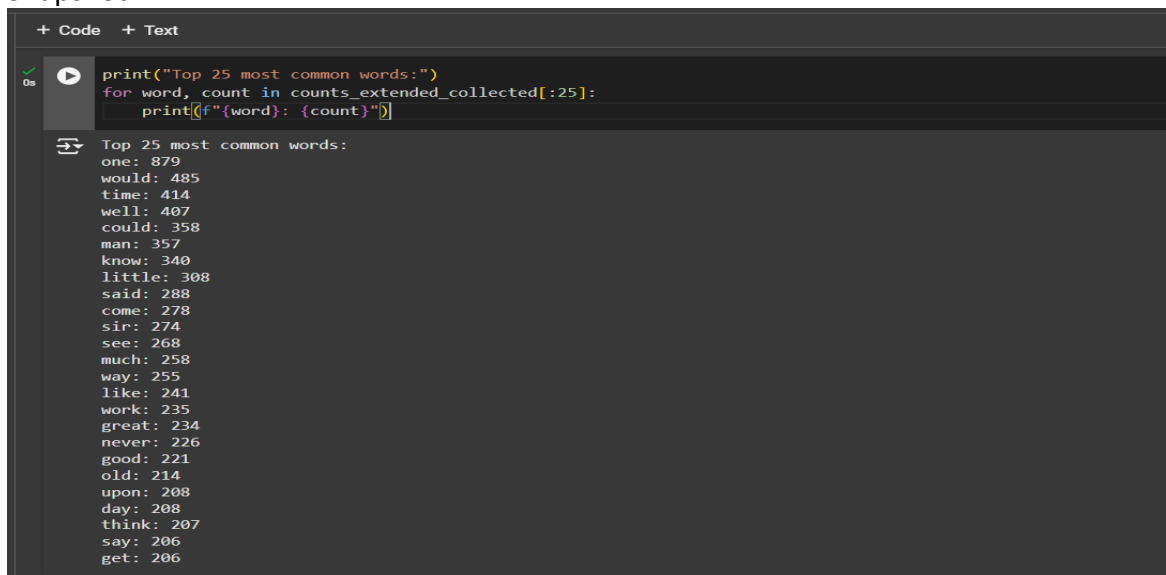
```
Output saved as output_1_extended.txt as list of key value pairs

[('one': 879), ('would': 485), ('time': 414), ('well': 407), ('could': 358), ('man': 357), ('know': 340), ('little': 308), ('said': 288), ('come': 278), ('sir': 274),
```

Analysis:

a. The code snippet prints the top 25 most frequent words from the sorted list of word counts. Since the words are already sorted in descending order, it directly displays them in "word: count" format.

Snapshot:



The screenshot shows a Jupyter Notebook interface with a code editor containing the following Python code:

```
print("Top 25 most common words:")
for word, count in counts_extended_collected[:25]:
    print(f"{word}: {count}")
```

The output of the code is displayed below the code cell:

```
Top 25 most common words:
one: 879
would: 485
time: 414
well: 407
could: 358
man: 357
know: 340
little: 308
said: 288
come: 278
sir: 274
see: 268
much: 258
way: 255
like: 241
work: 235
great: 234
never: 226
good: 221
old: 214
upon: 208
day: 208
think: 207
say: 206
get: 206
```

b. Spark breaks down jobs into multiple stages based on wide transformations like `reduceByKey`, `sortBy`, or `partitionBy`. These operations require shuffling data between different partitions, which introduces a boundary between stages. Narrow transformations (like `map` or `filter`) can be executed within a single stage because they don't require shuffling.

In this case:

- The union operation combines two RDDs (from two text files) in Stage 6.
- The `partitionBy` and `mapPartitions` operations are distributed across Stages 7 and 8, indicating that Spark needed to shuffle data between partitions, resulting in separate stages.

Snapshots of DAG visualizations:

1. Number of Jobs

Spark Jobs (?)

User: root
Total Uptime: 4.8 min
Scheduling Mode: FIFO
Completed Jobs: 4

▶ Event Timeline


▼ Completed Jobs (4)

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

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	collect at <ipython-input-167-4165b9ae064f>:1 collect at <ipython-input-167-4165b9ae064f>:1	2024/11/12 22:18:36	0.7 s	2/2 (1 skipped)	4/4 (2 skipped)
2	sortBy at <ipython-input-166-80c921acd931>:4 sortBy at <ipython-input-166-80c921acd931>:4	2024/11/12 22:18:27	0.5 s	1/1 (1 skipped)	2/2 (2 skipped)
1	sortBy at <ipython-input-166-80c921acd931>:4 sortBy at <ipython-input-166-80c921acd931>:4	2024/11/12 22:18:25	2 s	2/2	4/4
0	collect at <ipython-input-163-86a92e5f6141>:1 collect at <ipython-input-163-86a92e5f6141>:1	2024/11/12 22:18:08	2 s	2/2	4/4

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

2. Number of Stages

 3.5.3 Jobs Stages Storage Environment Executors WordCountCode application UI

Stages for All Jobs

Completed Stages: 7
Skipped Stages: 2

▼ Completed Stages (7)

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

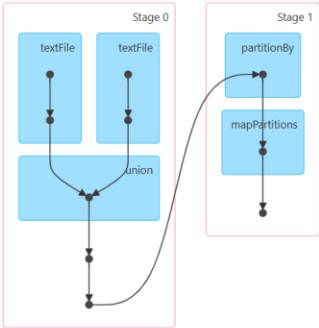
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
8	collect at <ipython-input-167-4165b9ae064f>:1 +details	2024/11/12 22:18:37	0.4 s	2/2			143.0 KiB	
7	sortBy at <ipython-input-166-80c921acd931>:4 +details	2024/11/12 22:18:36	0.3 s	2/2			162.9 KiB	143.0 KiB
5	sortBy at <ipython-input-166-80c921acd931>:4 +details	2024/11/12 22:18:27	0.5 s	2/2			162.9 KiB	
3	sortBy at <ipython-input-166-80c921acd931>:4 +details	2024/11/12 22:18:26	0.5 s	2/2			162.9 KiB	
2	reduceByKey at <ipython-input-166-80c921acd931>:10 +details	2024/11/12 22:18:25	1 s	2/2	896.9 KiB			162.9 KiB
1	collect at <ipython-input-163-86a92e5f6141>:1 +details	2024/11/12 22:18:09	0.3 s	2/2			255.6 KiB	
0	reduceByKey at <ipython-input-162-4145a6a19229>:6 +details	2024/11/12 22:18:08	1 s	2/2	896.9 KiB			255.6 KiB

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

Details for Job 0

Status: SUCCEEDED
Submitted: 2024/11/12 22:18:08
Duration: 2 s
Completed Stages: 2

- Event Timeline
- DAG Visualization

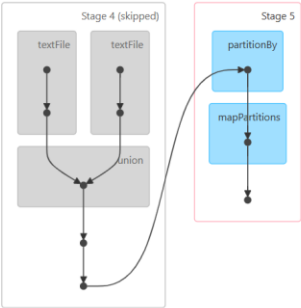


Completed Stages (2)

Details for Job 2

Status: SUCCEEDED
Submitted: 2024/11/12 22:18:27
Duration: 0.5 s
Completed Stages: 1
Skipped Stages: 1

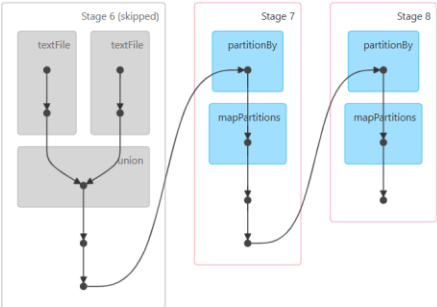
- Event Timeline
- DAG Visualization



Details for Job 3

Status: SUCCEEDED
Submitted: 2024/11/12 22:18:36
Duration: 0.7 s
Completed Stages: 2
Skipped Stages: 1

- Event Timeline
- DAG Visualization



Completed Stages (2)

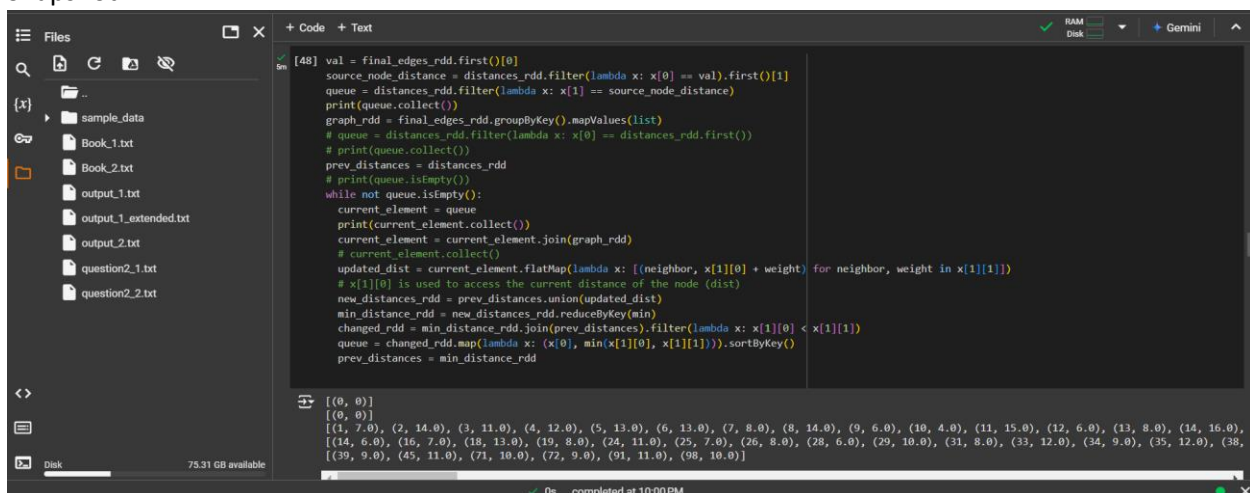
Part-2 Implement and analyze Dijkstra's Shortest Path algorithm.

Dijkstra's shortest path algorithm is a well-known graph traversal technique used to find the shortest path between a source node and all other nodes in a weighted graph. This implementation leverages PySpark to efficiently handle large-scale graph data and compute shortest paths in a distributed manner.

The code begins by reading edge data from two input text files (question2_1.txt and question2_2.txt) and parses the data into key-value pairs representing nodes and their respective edge weights. The edges are then combined using the union operation and added the weights which have same key's and sorted to ensure consistent processing. The algorithm initializes the source node distance to zero and sets all other nodes' distances to infinity. This allows the algorithm to iteratively compute the shortest paths by updating the distances of neighboring nodes as it traverses the graph.

The core of the algorithm works by processing each node in the queue, calculating the shortest distance to its neighbors, and updating the queue as necessary. The queue is dynamically modified to include only those nodes whose distances have changed. Once the algorithm completes, the final distances from the source node to all other nodes are saved in the output file output_2.txt.

Snapshot:



The screenshot displays a PySpark IDE interface. On the left, a file explorer shows a directory structure with files like 'sample_data', 'Book_1.txt', 'Book_2.txt', 'output_1.txt', 'output_1_extended.txt', 'output_2.txt', 'question2_1.txt', and 'question2_2.txt'. The main editor area contains Scala code implementing Dijkstra's algorithm. The code reads edge data from 'question2_1.txt' and 'question2_2.txt', processes it into an RDD, and iteratively updates the shortest distances for each node. The output of the algorithm is displayed in the bottom right corner, showing a list of nodes and their shortest distances from the source node (0).

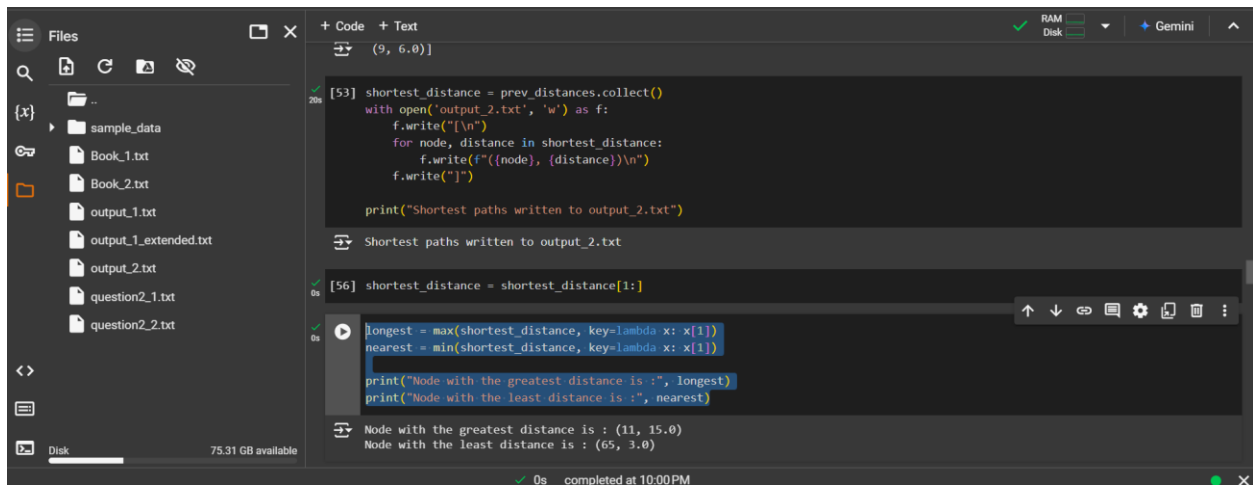
```
[48] val = final_edges_rdd.first()[0]
source_node_distance = distances_rdd.filter(lambda x: x[0] == val).first()[1]
queue = distances_rdd.filter(lambda x: x[1] == source_node_distance)
print(queue.collect())
graph_rdd = final_edges_rdd.groupByKey().mapValues(list)
# queue = distances_rdd.filter(lambda x: x[0] == distances_rdd.first())
# print(queue.collect())
prev_distances = distances_rdd
# print(queue.isEmpty())
while not queue.isEmpty():
    current_element = queue
    print(current_element.collect())
    current_element = current_element.join(graph_rdd)
    # current_element.collect()
    updated_dist = current_element.flatMap(lambda x: [(neighbor, x[1][0] + weight) for neighbor, weight in x[1][1]])
    # x[1][0] is used to access the current distance of the node (dist)
    new_distances_rdd = prev_distances.union(updated_dist)
    min_distance_rdd = new_distances_rdd.reduceByKey(min)
    changed_rdd = min_distance_rdd.join(prev_distances).filter(lambda x: x[1][0] < x[1][1])
    queue = changed_rdd.map(lambda x: (x[0], min(x[1][0], x[1][1]))).sortByKey()
    prev_distances = min_distance_rdd
```

Output:

```
[(0, 0)]
[(0, 0)]
[(1, 7.0), (2, 14.0), (3, 11.0), (4, 12.0), (5, 13.0), (6, 13.0), (7, 8.0), (8, 14.0), (9, 6.0), (10, 4.0), (11, 15.0), (12, 6.0), (13, 8.0), (14, 16.0),
(14, 6.0), (16, 7.0), (18, 13.0), (19, 8.0), (24, 11.0), (25, 7.0), (26, 8.0), (28, 6.0), (29, 10.0), (31, 8.0), (33, 12.0), (34, 9.0), (35, 12.0), (38,
(39, 9.0), (45, 11.0), (71, 10.0), (72, 9.0), (91, 11.0), (98, 10.0)]
```

0s completed at 10:00 PM

a) The output shows that the node with the greatest distance from the source node is node 11, with a distance of 15.0, while the node with the least distance is node 65, with a distance of 3.0. These results highlight the nodes that are the farthest and closest from the starting point, respectively.



```
[53] shortest_distance = prev_distances.collect()
with open('output_2.txt', 'w') as f:
    f.write("\n")
    for node, distance in shortest_distance:
        f.write(f'({node}, {distance})\n')
    f.write("\n")

print("Shortest paths written to output_2.txt")

[56] shortest_distance = shortest_distance[1:]

longest = max(shortest_distance, key=lambda x: x[1])
nearest = min(shortest_distance, key=lambda x: x[1])

print("Node with the greatest distance is :", longest)
print("Node with the least distance is :", nearest)

Node with the greatest distance is : (11, 15.0)
Node with the least distance is : (65, 3.0)
```

b) The execution is broken up into 492 stages in total, as indicated by the DAG visualization and the Spark WebUI screenshots. Here's a breakdown of how this number is derived:

Completed Stages: 63

Skipped Stages: 430

- The DAG visualization shows multiple stages that are marked as "skipped." These stages are skipped due to Spark's optimization techniques like stage reuse and caching, where Spark avoids redundant recomputation.
- The Spark WebUI confirms that many tasks were skipped across various stages (as seen in the "Tasks" column), further supporting that a large portion of the stages were optimized out during execution.

The stages correspond to the logical execution plan in Spark, and each stage typically represents a set of operations that can be computed in parallel within the available partitions. The shuffling of data between partitions, required by wide transformations, often leads to more stages since it involves network communication and disk I/O.

The DAG visualization from Spark's WebUI provides a detailed view of the job execution plan, showing how Spark distributed tasks into stages and handled data shuffling, which is key to understanding the performance and optimizations in the job execution.

Snapshots:

```
# https://dashboard.ngrok.com/auth
authtoken = "2odaNVr4bHYfaumRhkJVaB01GUS_7fijEi5bsKRslQQaENdKA"
ngrok.set_auth_token(authtoken)
spark = SparkSession.builder.master("local[*]").appName("DijkstraShortestPathCode").getOrCreate()
public_url = ngrok.connect(addr='http://localhost:4040')
print(f"Spark UI available at: {public_url}")

WARNING:pyngrok.process.ngrok:t=2024-11-13T03:25:25+0000 lvl=warn msg="can't bind default web address, trying alternatives" obj=we
Spark UI available at: NgrokTunnel: "https://13eb-35-234-0-59.ngrok-free.app" -> "http://localhost:4040"

[ ] sc.stop()
```

Total Number of Stages

Spark

3.5.3

Jobs

Stages

Storage

Environment

Executors

SQL / DataFrame

DijkstraShortestPathCode application UI

Stages for All Jobs

Completed Stages: 63
Skipped Stages: 430

Completed Stages (63)

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
492	collect at <ipython-input-53-821f8cb0e1b0>:1	+details 2024/11/13 02:53:49	20 s	127/127			21.2 KiB	
473	collect at <ipython-input-52-1136a8cb8f18>:1	+details 2024/11/13 02:53:00	20 s	127/127			21.2 KiB	
454	collect at <ipython-input-51-aecaf27d4153>:1	+details 2024/11/13 02:52:01	22 s	127/127			21.2 KiB	
435	collect at <ipython-input-50-bb23bbdd923e>:1	+details 2024/11/13 02:50:56	21 s	127/127			21.2 KiB	
416	runJob at PythonRDD.scala:181	+details 2024/11/13 02:45:21	3 s	13/13			856.0 B	
396	runJob at PythonRDD.scala:181	+details 2024/11/13 02:45:20	0.8 s	4/4			234.0 B	
376	runJob at PythonRDD.scala:181	+details 2024/11/13 02:45:20	0.1 s	1/1			68.0 B	
375	sortByKey at <ipython-input-49-e4fbbec79c8>:4	+details 2024/11/13 02:45:00	20 s	127/127			21.2 KiB	7.6 KiB
356	sortByKey at <ipython-input-49-e4fbbec79c8>:4	+details 2024/11/13 02:44:39	20 s	127/127			21.2 KiB	
337	sortByKey at <ipython-input-49-e4fbbec79c8>:4	+details 2024/11/13 02:44:18	21 s	127/127			21.2 KiB	
318	runJob at PythonRDD.scala:181	+details 2024/11/13 02:40:45	8 s	54/54				
298	runJob at PythonRDD.scala:181	+details 2024/11/13 02:40:30	15 s	100/100			11.9 KiB	
278	runJob at PythonRDD.scala:181	+details 2024/11/13 02:40:27	3 s	20/20			3.2 KiB	

Spark Jobs (?)

User: root
Total Uptime: 1.0 h
Scheduling Mode: FIFO
Completed Jobs: 41

Event Timeline

Completed Jobs (41)

Page: 1

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

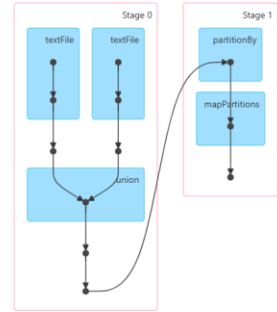
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
40	collect at <ipython-input-53-821f8cb0e1b0>:1 collect at <ipython-input-53-821f8cb0e1b0>:1	2024/11/13 02:53:49	20 s	1/1 (18 skipped)	127/127 (566 skipped)
39	collect at <ipython-input-52-1136a8cb8f18>:1 collect at <ipython-input-52-1136a8cb8f18>:1	2024/11/13 02:53:00	20 s	1/1 (18 skipped)	127/127 (566 skipped)
38	collect at <ipython-input-51-aecaf27d4153>:1 collect at <ipython-input-51-aecaf27d4153>:1	2024/11/13 02:52:01	22 s	1/1 (18 skipped)	127/127 (566 skipped)
37	collect at <ipython-input-50-bb23bbbd923e>:1 collect at <ipython-input-50-bb23bbbd923e>:1	2024/11/13 02:50:56	21 s	1/1 (18 skipped)	127/127 (566 skipped)
36	runJob at PythonRDD.scala:181 runJob at PythonRDD.scala:181	2024/11/13 02:45:21	3 s	1/1 (19 skipped)	13/13 (693 skipped)
35	runJob at PythonRDD.scala:181 runJob at PythonRDD.scala:181	2024/11/13 02:45:20	0.8 s	1/1 (19 skipped)	4/4 (693 skipped)

Details for Job 0

Status: SUCCEEDED
Submitted: 2024/11/13 02:34:49
Duration: 2 s
Completed Stages: 2

Event Timeline

DAG Visualization

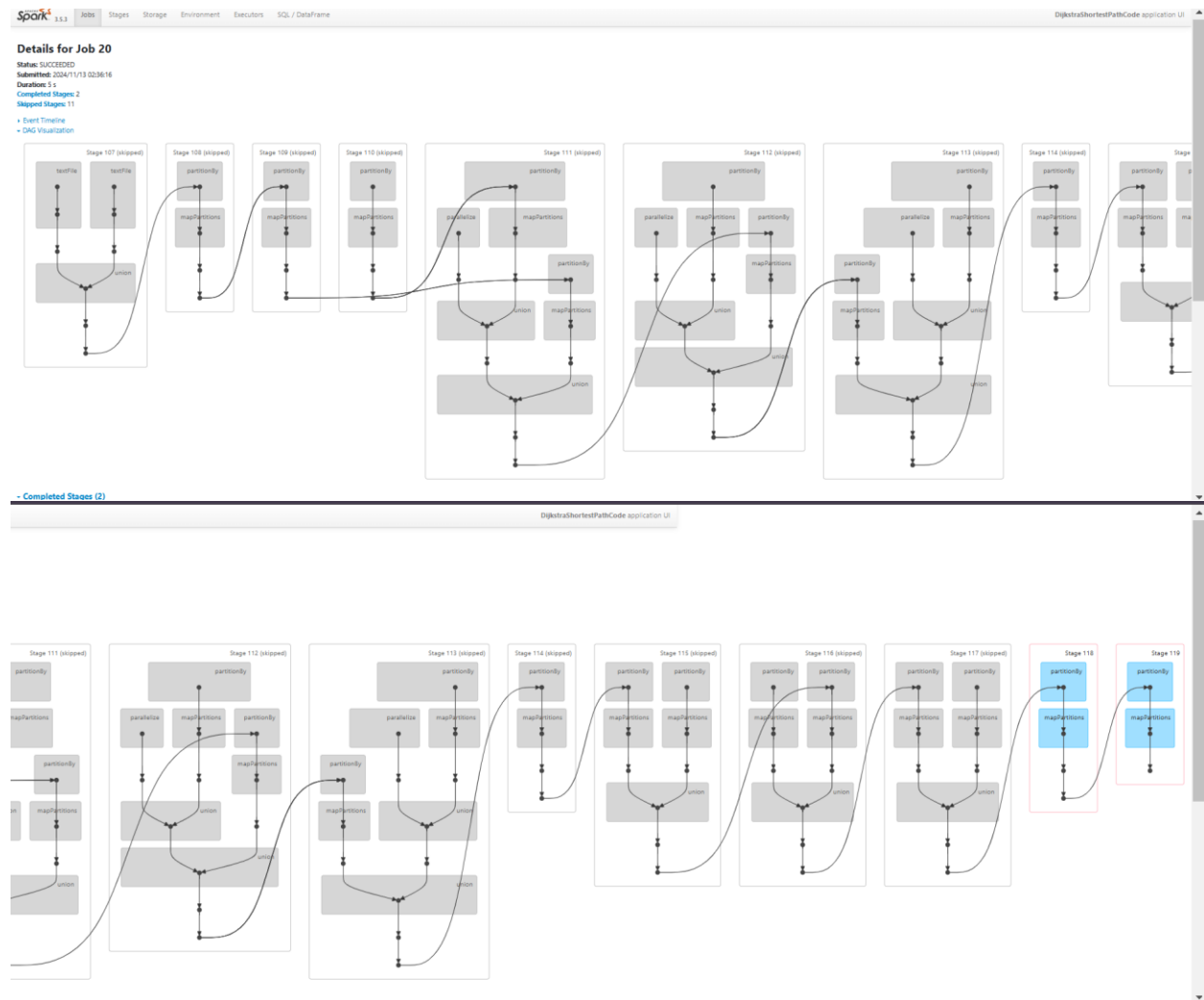


Completed Stages (2)

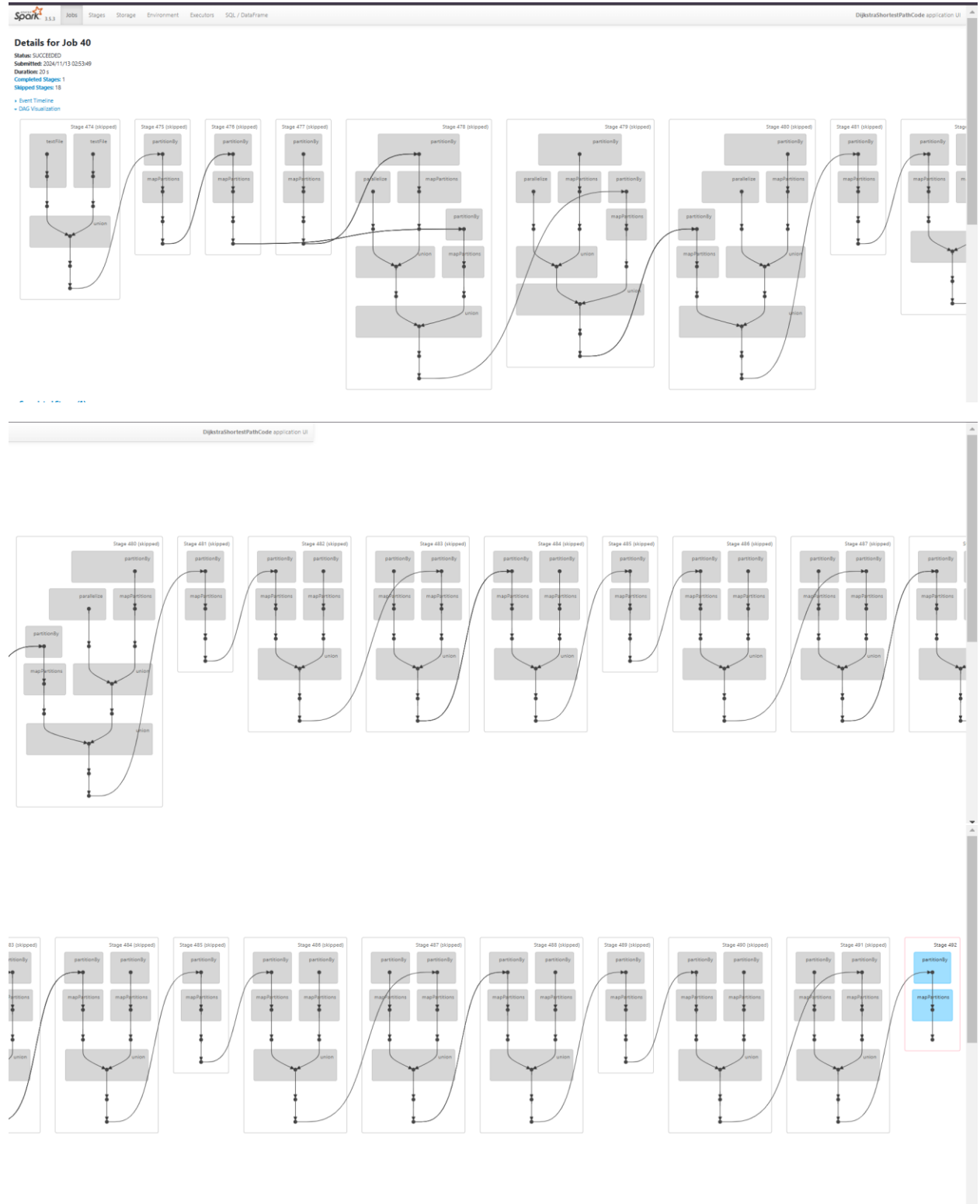
Page: 1

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

Mid



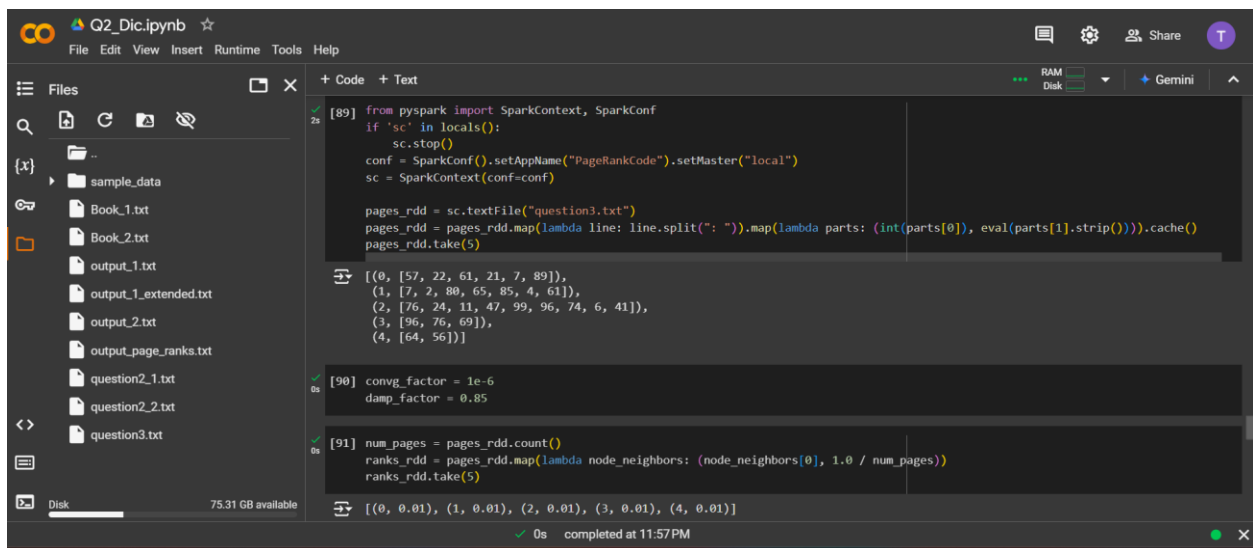
Last



Part-3 Implement and analyze Page-rank algorithm.

The PageRank algorithm is a ranking system used to evaluate the importance of each page in a network, based on the links (hyperlinks) between them. This implementation calculates the PageRank for all webpages in a simulated network of 100 pages. It uses an iterative approach where each page's rank is updated by considering the ranks of the pages linking to it.

The algorithm starts by reading the network structure from the question3.txt file and initializing each page's rank to a uniform value (1/number of pages). It then iterates over the network, propagating rank values based on the damping factor and link structure until the rank differences between iterations fall below a predefined convergence threshold (1e-6). The damping factor (0.85) helps prevent ranks from being distributed too evenly.



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files like Book_1.txt, Book_2.txt, output_1.txt, output_1_extended.txt, output_2.txt, output_page_ranks.txt, question2_1.txt, question2_2.txt, and question3.txt. The code editor contains the following code:

```
[89] from pyspark import SparkContext, SparkConf
if 'sc' in locals():
    sc.stop()
conf = SparkConf().setAppName("PageRankCode").setMaster("local")
sc = SparkContext(conf=conf)

pages_rdd = sc.textFile("question3.txt")
pages_rdd = pages_rdd.map(lambda line: line.split(":")).map(lambda parts: (int(parts[0]), eval(parts[1].strip()))).cache()
pages_rdd.take(5)

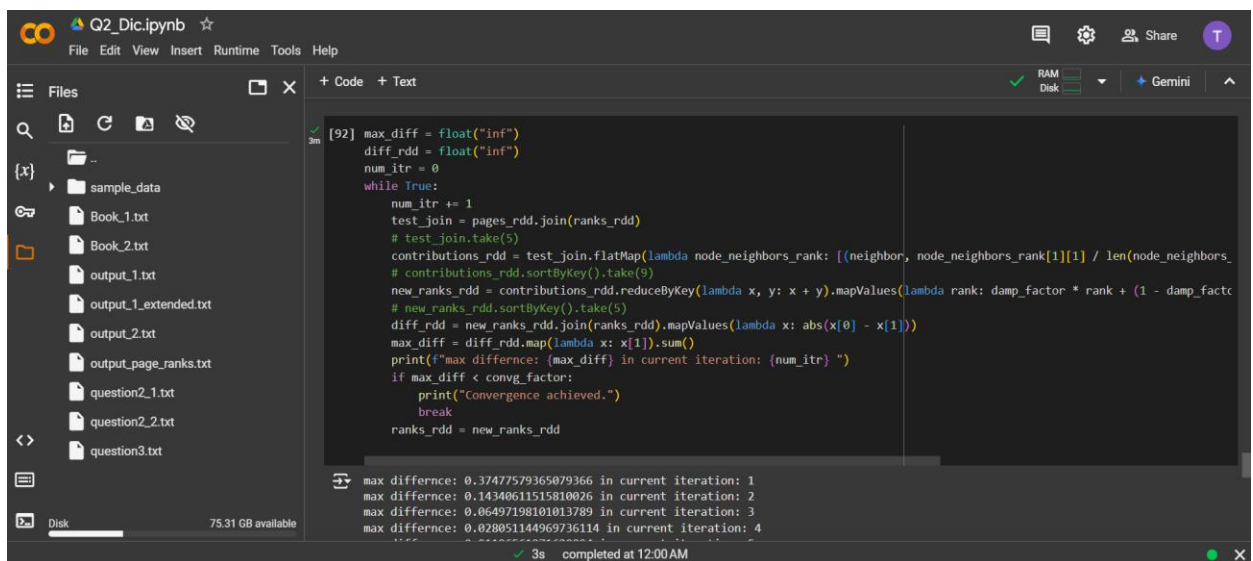
[(0, [57, 22, 61, 21, 7, 89]),
 (1, [7, 2, 80, 65, 85, 4, 61]),
 (2, [76, 24, 11, 47, 99, 96, 74, 6, 41]),
 (3, [96, 76, 69]),
 (4, [64, 56])]

[90] convg_factor = 1e-6
damp_factor = 0.85

[91] num_pages = pages_rdd.count()
ranks_rdd = pages_rdd.map(lambda node_neighbors: (node_neighbors[0], 1.0 / num_pages))
ranks_rdd.take(5)

[(0, 0.01), (1, 0.01), (2, 0.01), (3, 0.01), (4, 0.01)]
```

The code is executed, and the output shows the first 5 lines of the network structure and the initial ranks for the first 5 nodes.

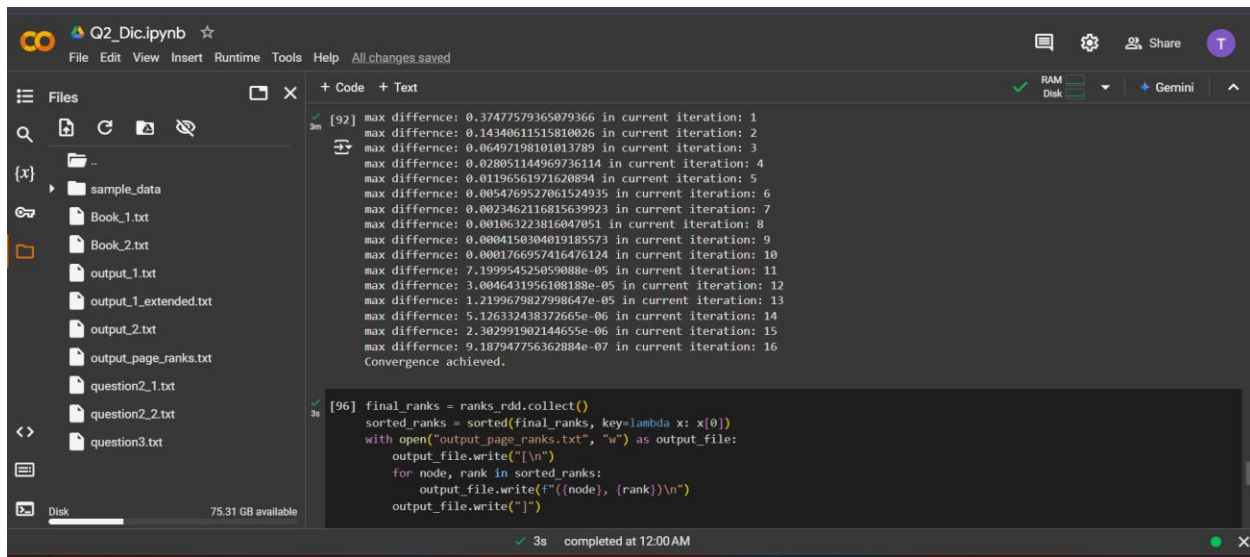


The screenshot shows the same Jupyter Notebook interface, but with the iterative PageRank algorithm code. The code is as follows:

```
[92] max_diff = float("inf")
diff_rdd = float("inf")
num_itr = 0
while True:
    num_itr += 1
    test_join = pages_rdd.join(ranks_rdd)
    # test_join.take(5)
    contributions_rdd = test_join.flatMap(lambda node_neighbors_rank: [(neighbor, node_neighbors_rank[1][1] / len(node_neighbors_rank[0]))])
    # contributions_rdd.sortByKey().take(9)
    new_ranks_rdd = contributions_rdd.reduceByKey(lambda x, y: x + y).mapValues(lambda rank: damp_factor * rank + (1 - damp_factor))
    diff_rdd = new_ranks_rdd.join(ranks_rdd).mapValues(lambda x: abs(x[0] - x[1]))
    max_diff = diff_rdd.map(lambda x: x[1]).sum()
    print(f"max difference: {max_diff} in current iteration: {num_itr}")
    if max_diff < convg_factor:
        print("Convergence achieved.")
        break
    ranks_rdd = new_ranks_rdd

max difference: 0.37477579365079366 in current iteration: 1
max difference: 0.14340611515810026 in current iteration: 2
max difference: 0.06497198101013789 in current iteration: 3
max difference: 0.028851144969736114 in current iteration: 4
```

The code is executed, and the output shows the maximum difference between ranks for each iteration, indicating convergence after 4 iterations.

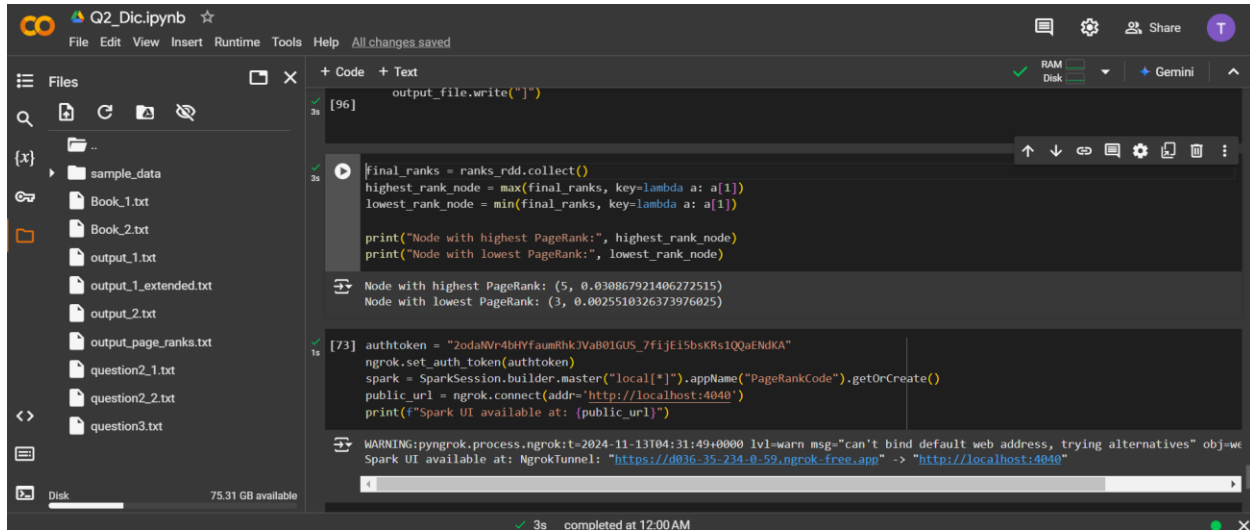


```
[92] max difference: 0.37477579365079366 in current iteration: 1
max difference: 0.14340611515810026 in current iteration: 2
max difference: 0.06497198101013789 in current iteration: 3
max difference: 0.028851144969736114 in current iteration: 4
max difference: 0.01196561971620894 in current iteration: 5
max difference: 0.0054769527061524935 in current iteration: 6
max difference: 0.0023462116815639923 in current iteration: 7
max difference: 0.001063223816047051 in current iteration: 8
max difference: 0.0004150304019185573 in current iteration: 9
max difference: 0.0001766957416476124 in current iteration: 10
max difference: 7.199954525059088e-05 in current iteration: 11
max difference: 3.0046431956108188e-05 in current iteration: 12
max difference: 1.2199679827998647e-05 in current iteration: 13
max difference: 5.126332438372665e-06 in current iteration: 14
max difference: 2.302991902144655e-06 in current iteration: 15
max difference: 9.187947756362884e-07 in current iteration: 16
Convergence achieved.

[96] final_ranks = ranks_rdd.collect()
sorted_ranks = sorted(final_ranks, key=lambda x: x[0])
with open("output_page_ranks.txt", "w") as output_file:
    output_file.write("\n")
    for node, rank in sorted_ranks:
        output_file.write(f"({node}, {rank})\n")
    output_file.write("\n")
```

The code prints the iteration number and the maximum difference between PageRank values in each iteration until convergence is reached.

After convergence, the node with the highest and lowest PageRank values is identified.
Highest Rank node is 5 and lowest rank node is 3.



```
[96] output_file.write("\n")

[73] final_ranks = ranks_rdd.collect()
highest_rank_node = max(final_ranks, key=lambda a: a[1])
lowest_rank_node = min(final_ranks, key=lambda a: a[1])

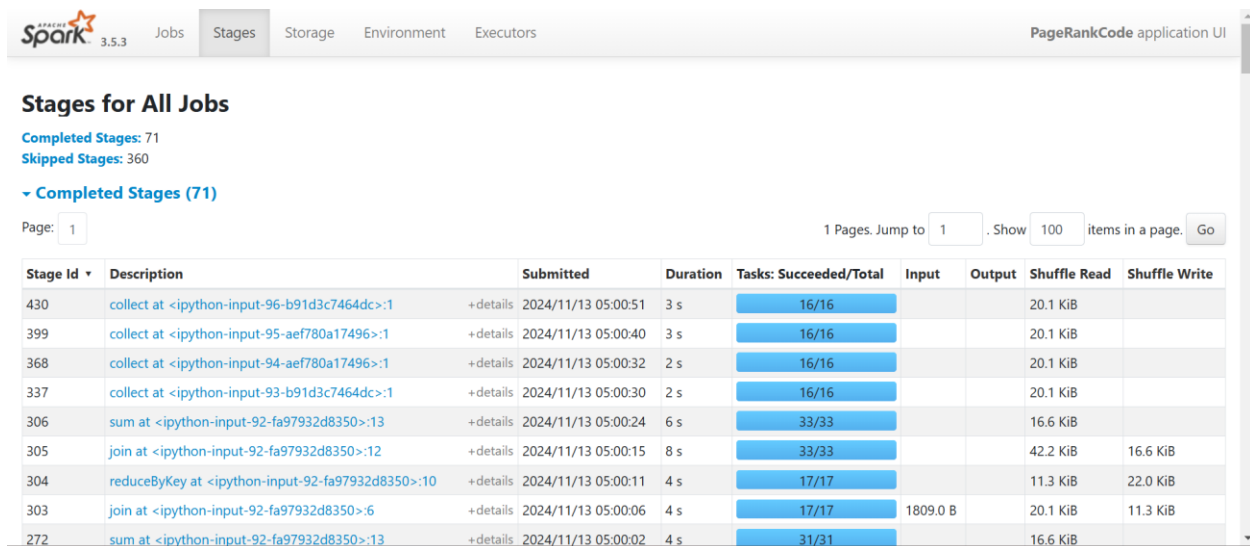
print("Node with highest PageRank:", highest_rank_node)
print("Node with lowest PageRank:", lowest_rank_node)

Node with highest PageRank: (5, 0.030867921406272515)
Node with lowest PageRank: (3, 0.0025510326373976025)

[73] authtoken = "2odaNvr4bHYfaumRhkJVaB01GUS_7fiJE15bsKRslQQaENdKA"
ngrok.set_auth_token(authtoken)
spark = SparkSession.builder.master("local[*]").appName("PageRankCode").getOrCreate()
public_url = ngrok.connect(addr='http://localhost:4040')
print(f"Spark UI available at: {public_url}")

WARNING:pyngrok.process.ngrok:t=2024-11-13T04:31:49+0000 lvl=warn msg="can't bind default web address, trying alternatives" obj=we
Spark UI available at: NgrokTunnel: "https://d036-35-234-0-59.ngrok-free.app" -> "http://localhost:4040"
```

The execution of this PageRank algorithm is broken up into multiple stages across several jobs



Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
430	collect at <ipython-input-96-b91d3c7464dc>:1	2024/11/13 05:00:51	3 s	16/16			20.1 KiB	
399	collect at <ipython-input-95-aef780a17496>:1	2024/11/13 05:00:40	3 s	16/16			20.1 KiB	
368	collect at <ipython-input-94-aef780a17496>:1	2024/11/13 05:00:32	2 s	16/16			20.1 KiB	
337	collect at <ipython-input-93-b91d3c7464dc>:1	2024/11/13 05:00:30	2 s	16/16			20.1 KiB	
306	sum at <ipython-input-92-fa97932d8350>:13	2024/11/13 05:00:24	6 s	33/33			16.6 KiB	
305	join at <ipython-input-92-fa97932d8350>:12	2024/11/13 05:00:15	8 s	33/33			42.2 KiB	16.6 KiB
304	reduceByKey at <ipython-input-92-fa97932d8350>:10	2024/11/13 05:00:11	4 s	17/17			11.3 KiB	22.0 KiB
303	join at <ipython-input-92-fa97932d8350>:6	2024/11/13 05:00:06	4 s	17/17	1809.0 B		20.1 KiB	11.3 KiB
272	sum at <ipython-input-92-fa97932d8350>:13	2024/11/13 05:00:02	4 s	31/31			16.6 KiB	

The execution of the PageRank algorithm in this Spark application is broken up into 71 completed stages, with 360 stages being skipped. The large number of stages reflects the complex and iterative nature of the PageRank computation. Each stage in the DAG (Directed Acyclic Graph) corresponds to a sequence of transformations or actions applied to the RDDs (Resilient Distributed Datasets).

Reasons for Multiple Stages such as

- **Iterative Algorithm:** The PageRank algorithm runs in a while loop until convergence. Each iteration creates new stages for the various transformations.
- **Multiple Transformations:** Within each iteration, there are several transformations like join, flatMap, reduceByKey, and mapValues. Each of these, especially the wide transformations like join and reduceByKey, typically result in new stages.
- **Actions Triggering Jobs:** Actions like take(5) and count() trigger new jobs, each potentially consisting of multiple stages.
- **Caching:** The cache() operation on pages_rdd creates an additional stage to persist the data.
- **Shuffle Operations:** Operations like join and reduceByKey involve data shuffling, which necessitates stage boundaries.

The screenshot of the DAG will illustrate the breakdown of stages, showcasing the transformation sequence and the interdependencies among them.

SPARK
3.5.3

JobsStagesStorageEnvironmentExecutors

PageRankCode application UI

Spark Jobs ^(?)

User: root
Total Uptime: 15 min
Scheduling Mode: FIFO
Completed Jobs: 23

▶ Event Timeline

▼ Completed Jobs (23)

Page: 1

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

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
22	collect at <ipython-input-96-b91d3c7464dc>:1 collect at <ipython-input-96-b91d3c7464dc>:1	2024/11/13 05:00:51	3 s	1/1 (30 skipped)	16/16 (270 skipped)
21	collect at <ipython-input-95-aef780a17496>:1 collect at <ipython-input-95-aef780a17496>:1	2024/11/13 05:00:40	3 s	1/1 (30 skipped)	16/16 (270 skipped)
20	collect at <ipython-input-94-aef780a17496>:1 collect at <ipython-input-94-aef780a17496>:1	2024/11/13 05:00:32	2 s	1/1 (30 skipped)	16/16 (270 skipped)
19	collect at <ipython-input-93-b91d3c7464dc>:1 collect at <ipython-input-93-b91d3c7464dc>:1	2024/11/13 05:00:30	2 s	1/1 (30 skipped)	16/16 (270 skipped)

SPARK
3.5.3

JobsStagesStorageEnvironmentExecutors

PageRankCode application UI

Details for Job 0

Status: SUCCEEDED
Submitted: 2024/11/13 04:57:06
Duration: 1 s
Completed Stages: 1

▶ Event Timeline

▼ DAG Visualization

The DAG visualization for Job 0 shows a single stage, Stage 0, which is highlighted with a red border. Inside Stage 0, there is a blue box labeled 'textFile'. Below the 'textFile' box, there is a vertical sequence of nodes: a black dot, a green dot, and another black dot, connected by lines.

SPARK
3.5.3

JobsStagesStorageEnvironmentExecutors

PageRankCode application UI

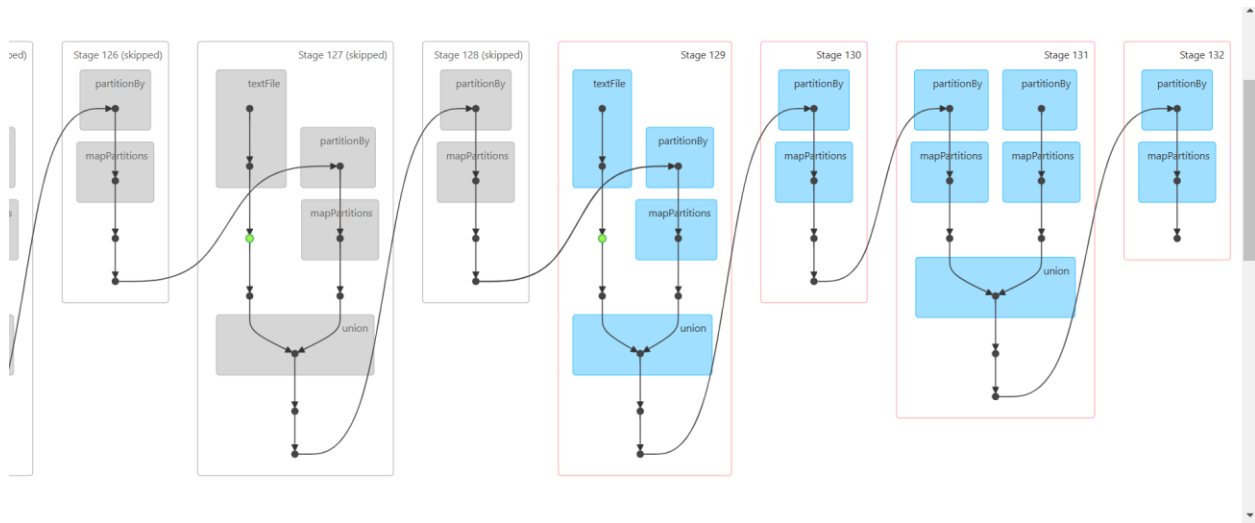
Details for Job 12


Status: SUCCEEDED
Submitted: 2024/11/13 04:58:18
Duration: 14 s
Completed Stages: 4
Skipped Stages: 18

▶ Event Timeline

▼ DAG Visualization

The DAG visualization for Job 12 shows a sequence of stages. Stages 111, 112, 113, 114, 115, 116, 117, and 118 are all marked as 'skipped'. Each skipped stage contains a 'textFile' box. Between the skipped stages, there are 'partitionBy' and 'mapPartitions' boxes. The DAG also shows 'union' operations connecting the stages. The visualization is a complex graph showing the flow of data through the stages and operations.



 3.5.3

JobsStagesStorageEnvironmentExecutors

PageRankCode application UI

Details for Job 22

Status: SUCCEEDED
Submitted: 2024/11/13 05:00:51
Duration: 3 s
Completed Stages: 1
Skipped Stages: 30

[Event Timeline](#)
[DAG Visualization](#)

This diagram shows a portion of a Spark DAG for stages 400 through 407. All stages are marked as 'skipped' and are shown in grey boxes. Stage 400 contains a 'textFile' node, a 'partitionBy' node, a 'mapPartitions' node, and a 'union' node. Stage 401 contains a 'partitionBy' node and a 'mapPartitions' node. Stage 402 contains a 'textFile' node, a 'partitionBy' node, a 'mapPartitions' node, and a 'union' node. Stage 403 contains a 'partitionBy' node and a 'mapPartitions' node. Stage 404 contains a 'textFile' node, a 'partitionBy' node, a 'mapPartitions' node, and a 'union' node. Stage 405 contains a 'partitionBy' node and a 'mapPartitions' node. Stage 406 contains a 'textFile' node, a 'partitionBy' node, a 'mapPartitions' node, and a 'union' node. Stage 407 contains a 'partitionBy' node and a 'mapPartitions' node. The diagram illustrates the flow of data between these stages, with arrows indicating the sequence of operations.

