

Report

Assignment 3

Team - shaivipa-kritikku

Part1 Review paper on Hadoop/HDFS

We've studied an in-depth examination of HDFS, the core storage system for Apache Hadoop, presented in the research paper "The Hadoop Distributed File System" authored by Shvachko and colleagues. The writers, who work for Yahoo!, offer valuable insights into the deployment and management of large-scale HDFS clusters, drawing from their extensive experience.

In our review, we note that HDFS's fundamental concepts and goals are outlined in the paper's introduction, emphasizing its capacity to store and process enormous datasets across numerous commodity servers. The authors discuss critical features like data replication (typically tripled), fault tolerance, and high-throughput data access, as well as the customizable 64 MB default block size and its effects on I/O performance and storage efficiency.

We found a detailed explanation of HDFS's master-slave architecture is provided, with the NameNode acting as the central metadata manager and DataNodes (up to 4,000 per cluster) handling data storage and serving. The paper elaborates on these components' roles, including the NameNode's file system namespace management (supporting up to 60 million files and 63 million blocks) and client access coordination, as well as the DataNodes' responsibilities in storing data blocks (typically 54,000 block replicas per node), fulfilling read and write requests, and conducting block replication and integrity checks using checksums.

We observed that the authors offer a thorough breakdown of data flow during file read and write operations, illustrating the interactions between clients, NameNode, and DataNodes. Detailed discussions cover the block placement policy, which takes into account factors like rack awareness and data locality, emphasizing the importance of minimizing inter-rack network traffic and enhancing fault tolerance. The replication pipeline, where data streams from the client to the first DataNode and then simultaneously to the second and third, is also described, with an explanation of how the pipeline is organized to minimize total network distance and how data is pushed in 64 KB packets.

In our analysis, we noted various HDFS features and optimizations are highlighted, including the block scanner for regular integrity checks (scanning all blocks in a large cluster each fortnight and finding about 20 bad replicas), the balancer tool for maintaining uniform data distribution, and the decommissioning process for safely removing nodes. Advanced topics such as snapshot support and the Hadoop Archive (HAR) file format are also addressed.

We found the paper shares practical experiences and challenges in operating large-scale HDFS clusters at Yahoo!, providing insights into hardware configurations (typical cluster node with 2 quad-core Xeon CPUs, 16 GB RAM, 4x1 TB SATA disks, 1 Gbps Ethernet), data volumes (25 PB of online data storage across 25,000 nodes), and operational aspects. The authors discuss resource allocation, quota management, and access control mechanisms, as well as how DataNodes communicate with the NameNode through periodic heartbeats (every 3 seconds) and block reports.

We were impressed by the benchmark results demonstrating HDFS's exceptional I/O throughput capabilities, including specific figures for DFSIO Read (66 MB/s per node), DFSIO Write (40 MB/s per node), Busy Cluster Read (1.02 MB/s per node), Busy Cluster Write (1.09 MB/s per node), and the Sort benchmark (1 TB sorted in 62 seconds on 1460 nodes, 1 PB sorted in 16.25 hours on 3658 nodes). The paper also mentions Yahoo!'s record-breaking performance in the GraySort competition.

We noted that the DistCp tool, which uses MapReduce for efficient inter-cluster and intra-cluster data copying, is introduced along with its benefits for managing large datasets across HDFS clusters.

In our review, we found that ongoing research and development efforts aimed at enhancing HDFS's scalability, availability, and performance are outlined, including plans to improve NameNode scalability (supporting up to 100 million files per NameNode),

introduce multiple namespace support, and develop more efficient inter-cluster collaboration techniques. The authors address NameNode scalability limitations and propose solutions such as federation and distributed NameNodes.

In conclusion, we find that "The Hadoop Distributed File System" by Shvachko et al. provides a comprehensive and technically detailed examination of HDFS, offering valuable guidance for understanding, deploying, and managing HDFS clusters in production environments. The authors' extensive experience and insights, backed by numerous performance metrics, system specifications, and operational statistics, make this paper an indispensable resource for those working with Hadoop and big data technologies.

Part2 Implement and analyze Dijkstra's Shortest Path algorithm

1. You must write a basic Dijkstra's shortest path algorithm for the text file provided as part of the assignment. Where the first column of each row is the initial node, the second column of each row is the destination, and the third column is the weight associated with the connection. The graph representation of the nodes from the question1.txt file is as follows:

Algorithm

SSSP-MapReduce(in_f, out_f)

1. Initialize Spark session:

- `spark ← SparkSession.builder.appName("SSSPMR").getOrCreate()`
- `graph_rdd ← None`

2. Define static method parse(line):

- `parts ← line.split(',')`
- `return (int(parts[0].strip()), (int(parts[1].strip()), float(parts[2].strip())))`

3. Define static method init_nodes(edges_rdd):

- Inner function create_node(n_id, edges):
 - `return (n_id, N(n_id, 0 if n_id == 0 else float('inf'), list(edges)))`
- `grouped_edges ← edges_rdd.groupByKey()`
- `return grouped_edges.map(lambda x: create_node(x[0], x[1]))`

4. Define class N(n_id, dist, adj):

- `self.n_id ← n_id`
- `self.dist ← dist`
- `self.adj ← adj`

5. Define class M:

- Static method map(n_id, node):
 - `yield (n_id, node)`
 - For each `(adj_n, w)` in `node.adj`:
 - `yield (adj_n, node.dist + w)`

6. Define class R:

- Static method reduce(n_id, vals):
 - `min_dist ← float('inf')`
 - `node ← None`
 - For each `val` in `vals`:
 - If `isinstance(val, N)`:
 - `node ← val`
 - Else:
 - `min_dist ← min(min_dist, val)`
 - If `node`:
 - `node.dist ← min(node.dist, min_dist)`
 - `return (n_id, node)`

7. Define method run_sssp():

- `iteration ← 0`
- While `True`:

```

- Print `Iteration {iteration}`
- `mapped <- graph_rdd.flatMap(lambda x: M.map(x[0], x[1]))`
- `new_graph <- mapped.groupByKey().map(lambda x: R.reduce(x[0], list(x[1])))`
- Define function compare_dists(old, new):
  - `return old.dist == new.dist`
- `unchanged <- graph_rdd.join(new_graph).map(lambda x: compare_dists(x[1][0], x[1][1])).reduce(lambda x, y: x and y)`
- If `unchanged`:
  - Break
- `graph_rdd <- new_graph`
- `iteration += 1`
- `return graph_rdd`

```

8. Define method run():

```

- Print `Spark Web UI available at: {spark.sparkContext.uiWebUrl}`
- `input_rdd <- spark.sparkContext.textFile(in_f)`
- `edges_rdd <- input_rdd.map(parse)`
- `graph_rdd <- init_nodes(edges_rdd)`
- `result_rdd <- run_sssp()`
- `results <- result_rdd.collect()`
- For each `(n_id, node)` in `sorted(results)`:
  - Print `Shortest distance to node {n_id}: {node.dist}`
- `dist_to_4 <- next(node.dist for n_id, node in results if n_id == 4)`
- Print `Shortest distance from node 0 to node 4: {dist_to_4}`
- With open `out_f` as `f`:
  - For each `(n_id, node)` in `sorted(results)`:
    - `f.write(f"\n{n_id},{node.dist}\n")`
- Print `Results written to {out_f}`
- Print `Keeping Spark context alive for 100 minutes. Access the Web UI now.`
- `time.sleep(6000)`
- `spark.stop()`

```

9. Main execution:

```

- `in_f <- "path/to/input/file"`
- `out_f <- "path/to/output/file"`
- `sssp <- SSSPMR(in_f, out_f)`
- `sssp.run()`

```

Algorithm Explanation

In designing this algorithm, we've implemented Dijkstra's Single Source Shortest Path (SSSP) algorithm using MapReduce, specifically adapting it for Spark.

First, for graph representation, we chose to use the edge list format we talked about in class. Each line in our input file represents an edge with the format: source_node, destination_node, weight. We created a `parse` method that converts this into an RDD of (source_node, (destination_node, weight)) tuples. This aligns with the graph representations we discussed.

For the node structure, we created an `N` class that's very similar to the Node object we mentioned in the slides. It contains the nodeId (which we called n_id), the distanceLabel (dist), and the adjacencyList (adj). This structure allows us to keep all the necessary information for each node in our graph.

Now, let's talk about how we implemented the iteration and MapReduce aspects. Remember how we discussed that each iteration in the algorithm would be a MapReduce job? That's exactly what we did here. The `run_sssp` method coordinates multiple MapReduce iterations, just like we talked about in class.

For the Mapper, we created an `M` class that closely resembles the Mapper class we saw in the slides. It emits two types of key-value pairs: `(nodeId, Node)` for the current node, and `(adjacentNodeId, updatedDistance)` for each adjacent node. This is how we propagate distance information through the graph.

The Reducer, which we implemented in the `R` class, is very similar to what we discussed in class. It finds the minimum distance among all received distances and updates the node's distance if a shorter path is found. This is crucial for implementing Dijkstra's algorithm in a distributed setting.

For termination, we used the condition we discussed in the slides: the algorithm stops when no distances change between iterations. This ensures we've found the shortest path to all reachable nodes.

In terms of how this implements Dijkstra's algorithm, we start with the source node (node 0) having a distance of 0, and all other nodes with infinite distance. In each iteration, the algorithm updates the distances of adjacent nodes, similar to how Dijkstra's algorithm expands the node with the shortest distance. The key difference is that by using MapReduce, this happens in parallel for multiple nodes.

One of the challenges we discussed in class was how to handle large graphs that don't fit in the memory of a single machine. By using Spark RDDs, our implementation can handle such large graphs, distributing the computation across a cluster.

In essence, what we've done here is adapt Dijkstra's algorithm to work in a distributed setting using MapReduce, allowing it to process large graphs efficiently. It follows the general structure and principles we outlined in the lecture slides while leveraging Spark's distributed computing capabilities. This approach allows us to tackle the kind of big data graph problems we discussed in class.

2. The algorithm should read the file, compute the shortest path between the first node indicated by 0 to the node indicated by 4 and print out the value. Additionally, save the distances to all the nodes in a text file named output_1.txt.

```
WARN: UserWarning: Please instance psutil to have better support with splitting  
Iteration 1  
Iteration 2  
Iteration 3  
Shortest distance to node 0: 0  
Shortest distance to node 1: 2.0  
Shortest distance to node 2: 3.0  
Shortest distance to node 3: 4.0  
Shortest distance to node 4: 3.0  
Shortest distance from node 0 to node 4: 3.0  
Results written to /Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstras/output/output_1  
.txt  
Keeping Spark context alive for 100 minutes. Access the Web UI now.  
24/07/02 18:10:30 WARN GarbageCollectionMetrics: To enable non-built-in garbage collector(s) List(G1 Concurrent GC), users s  
hould configure it(them) to spark.eventLog.gcMetrics.youngGenerationGarbageCollectors or spark.eventLog.gcMetrics.oldGenerat  
ionGarbageCollectors
```

3. Additionally, save the shortest distances to all the nodes in a text file named output_1.txt and provide a screenshot of the same.



4. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's Web UI.

The execution of the Single Source Shortest Path (SSSP) algorithm in PySpark is split into 27 stages, but not all of these stages are run. Here's why:

1. Total stages created: 27

2. Stages actually executed: 14

3. Stages skipped: 13

This large number of stages and the omission of certain stages can be attributed to the SSSP algorithm's iterative character and Spark's lazy evaluation:

1. Iterative Algorithm: The SSSP algorithm operates in iterations until it reaches convergence. Each iteration has the potential to generate new stages for its operations.

2. RDD Operations: The algorithm employs various RDD operations that can initiate stage creation:

groupByKey(): This operation induces a shuffle, resulting in the creation of a new stage.

join(): This also induces a shuffle and leads to the creation of a new stage.

union(): Although this doesn't induce a shuffle, it can contribute to the creation of a new stage when combined with other operations.

3. Lazy Evaluation: Spark employs lazy evaluation, which means it generates an execution plan (including all possible stages) prior to actual execution. This explains why we observe a total of 27 stages being created.

4. Dynamic Optimization: During runtime, Spark's DAG scheduler optimizes the execution. It may conclude that certain stages are unnecessary (e.g., if data from a previous iteration can be reused), resulting in stage skipping.

5. Convergence: The algorithm terminates when it converges (i.e., no changes in shortest paths). This implies that in the final iteration, some of the planned stages are not required and are therefore skipped.

6. Caching and Persistence: The utilization of caching or persistence of RDDs can influence which stages need to be recomputed in each iteration.

The code demonstrates a while loop that persists until convergence is achieved. In each iteration, it carries out:

A map operation (`flatMap`)

A reduce operation (`groupByKey` followed by `map`)

A join operation to check for convergence

Each of these operations, particularly `groupByKey` and `join`, can generate new stages. The precise number of stages executed is contingent upon the number of iterations required for the algorithm to converge.

The large number of stages is indicative of Spark's capability to decompose the computation into smaller, parallelizable units, facilitating distributed processing. However, not all stages are invariably essential, which is why Spark dynamically optimizes the execution by omitting unnecessary stages.

Spark UI DAG visualizations shows for a few stages

Starting Few stages

Stages 0, 1 and 2 are shown below

DAG visualization of Stage 0:



Primary Operation: groupByKey at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:27

Detailed Breakdown:

textFile:

- Utilized on two occasions to process
"/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/question1.txt"
- Objective: Ingests the input data, presumably containing edge details for the graph representation
- Implication: Duplication of data to ensure fault tolerance and resilience

PythonRDD [2]:

- Carries out the groupByKey operation to aggregate the data
- Goal: Utilizes custom Python logic to group the edge information based on the source nodes
- Implication: Potential performance impact due to the execution of Python code

PairwiseRDD [3]:

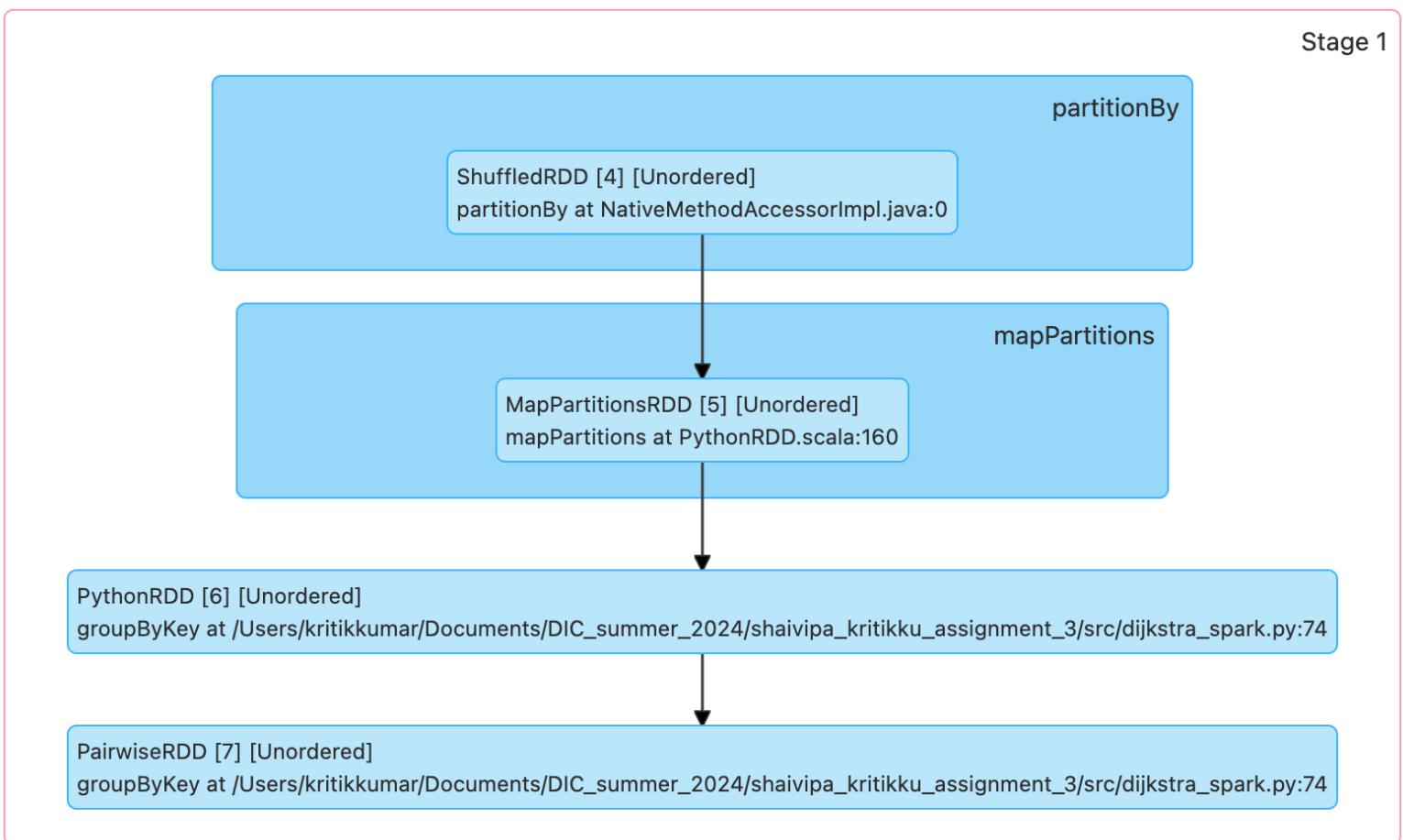
- Outcome of the groupByKey operation performed in the previous step
- Aim: Signifies the initial structure of the graph

- Implication: Consists of keyvalue pairs, where the keys represent source nodes and the values denote the corresponding adjacency lists

Analysis:

This initial stage constructs the graph structure without performing any shuffling operations, preserving the original partitioning of the data. The presence of duplicate textFile operations highlights an emphasis on ensuring data availability and fault tolerance. The utilization of PythonRDD suggests the application of custom preprocessing steps prior to the grouping operation, potentially involving the parsing of edge information or the initialization of distances. The resulting PairwiseRDD serves as the groundwork for the subsequent computational stages.

DAG visualization of Stage 1



Primary Operation: `groupByKey` at

`/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:74`

Detailed Breakdown:

1. `partitionBy`:

- Outcome: Generation of ShuffledRDD [4]
- Objective: Redistribution of data across the cluster nodes
- Consequence: Substantial network I/O overhead, but guarantees uniform data distribution for balanced processing

2. ShuffledRDD [4]:

- Originates from the `partitionBy` operation

- Role: Embodies the redistributed data following the shuffle process
- Implication: Optimal data placement achieved for subsequent computational stages

3. mapPartitions:

- Yields MapPartitionsRDD [5] as the output
- Function: Performs bulk transformations on individual partitions
- Benefit: Efficient localized processing, minimizing data transfer requirements

4. PythonRDD [6]:

- Responsible for executing the groupByKey operation
- Utilizes custom Python logic for grouping the data
- Advantage: Offers flexibility in defining grouping criteria
- Drawback: Potential performance impact due to Python execution overhead

5. PairwiseRDD [7]:

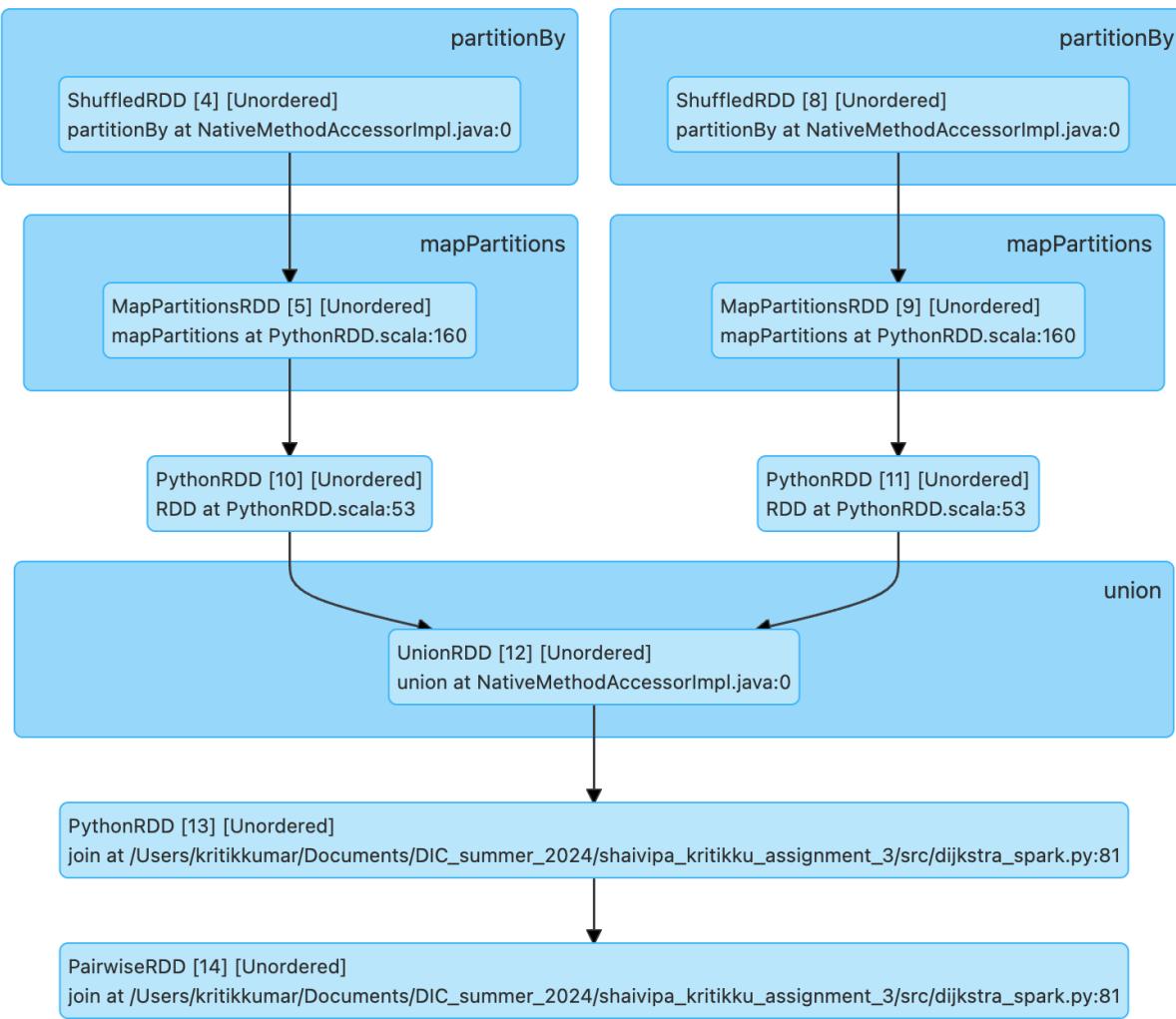
- Produced by the groupByKey operation
- Signifies the grouped data structure
- Readiness: Prepared for the subsequent processing stage

Analysis:

This particular stage encompasses a significant shuffling operation, which is essential for preparing the data for the initial iteration of the algorithm. The partitionBy step plays a crucial role in ensuring an even distribution of data across the cluster, thereby promoting optimal load balancing. The mapPartitions step enables efficient processing within each partition, potentially involving the initialization of distances or the preparation of data structures necessary for tracking paths. While the utilization of PythonRDD for the groupByKey operation offers flexibility, it may introduce some performance overhead associated with Python execution.

DAG visualization of Stage 2

Stage 2



Primary Operation: join at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:81

Detailed Breakdown:

1. Two parallel branches:

Branch 1:

a. ShuffledRDD [4]:

- Utilized from the previous Stage 1
- Objective: Supplies shuffled data for the join operation
- Benefit: Optimal data reuse, eliminating the need for redundant shuffling

b. mapPartitions:

- Yields MapPartitionsRDD [5]
- Purpose: Performs local transformations prior to the join
- Advantage: Optimizes data within each partition before the resourceintensive join operation

c. PythonRDD [10]:

- Carries out custom processing logic
- Goal: Prepares the data for the join operation using Python
- Tradeoff: Offers flexibility in preprocessing but may impact performance

2. Branch 2:

a. ShuffledRDD [8]:

- Introduces a new shuffle operation
- Aim: Redistributes the second dataset in preparation for the join
- Consequence: Guarantees data alignment for the join, but incurs network I/O overhead

b. mapPartitions:

- Generates MapPartitionsRDD [9]
- Function: Applies local transformations to the second dataset
- Benefit: Enables parallel optimization of the second dataset

c. PythonRDD [11]:

- Implements custom processing on the second dataset
- Objective: Prepares the second dataset for the join operation
- Advantage: Supports complex preprocessing logic

3. union:

- Results in the creation of UnionRDD [12]
- Purpose: Merges the two parallel branches
- Benefit: Prepares the datasets for joining without incurring additional shuffling overhead

4. PythonRDD [13]:

- Responsible for executing the join operation
- Utilizes custom join logic implemented in Python
- Tradeoff: Provides flexibility in join implementation but may impact performance

5. PairwiseRDD [14]:

- Represents the outcome of the join operation
- Holds the updated path information
- Significance: Contains the latest shortest path data

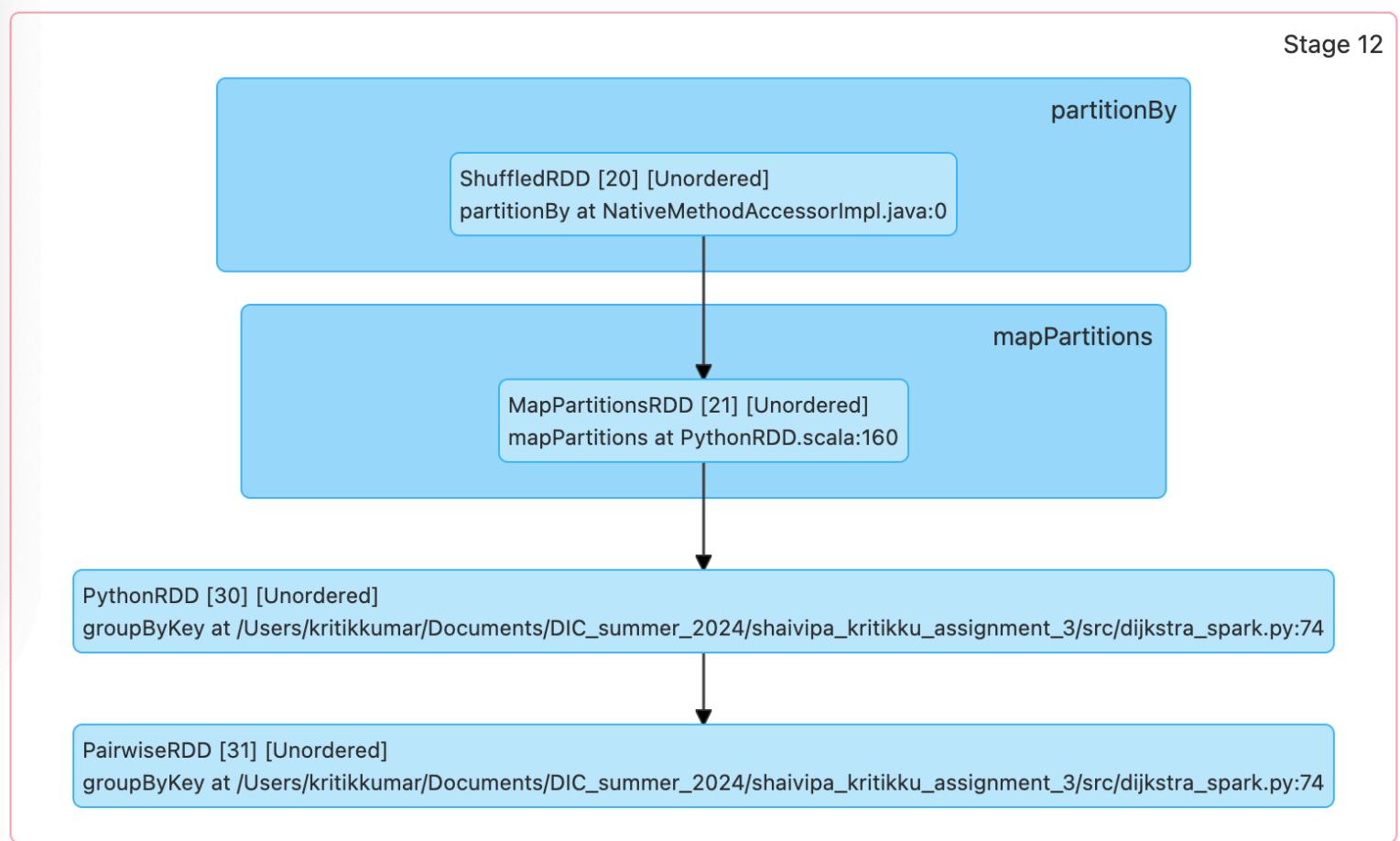
Analysis:

This intricate stage plays a pivotal role in performing a critical join operation, likely merging the current shortest paths with newly discovered paths. The utilization of two parallel branches, each with its own shuffle operation, ensures that both datasets are optimally distributed for the join. The mapPartitions operations within each branch facilitate efficient local preprocessing. The union operation combines these optimized datasets without necessitating further shuffling. The final join, implemented as a PythonRDD, offers flexibility in defining the join logic but may introduce some performance tradeoffs. This stage is crucial for updating the shortest paths by incorporating any newly discovered shorter routes.

Middle Stages

Stages 12, 13 and 14 are shown below

DAG visualization of Stage 12:



Primary Operation: groupByKey at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:74

Detailed Breakdown:

1. partitionBy:

- * Outcome: Generation of ShuffledRDD [20]
- * Objective: Redistributes the data based on the updated information
- * Consequence: Guarantees optimal data placement for the subsequent iteration

2. mapPartitions:

- * Yields MapPartitionsRDD [21]
- * Function: Performs transformations within the newly created partitions
- * Benefit: Enables efficient local processing of the reshuffled data

3. PythonRDD [30]:

- * Responsible for executing the groupByKey operation
- * Utilizes custom grouping logic implemented in Python
- * Tradeoff: Offers flexibility in grouping but may impact performance

4. PairwiseRDD [31]:

- * Represents the result of the groupByKey operation

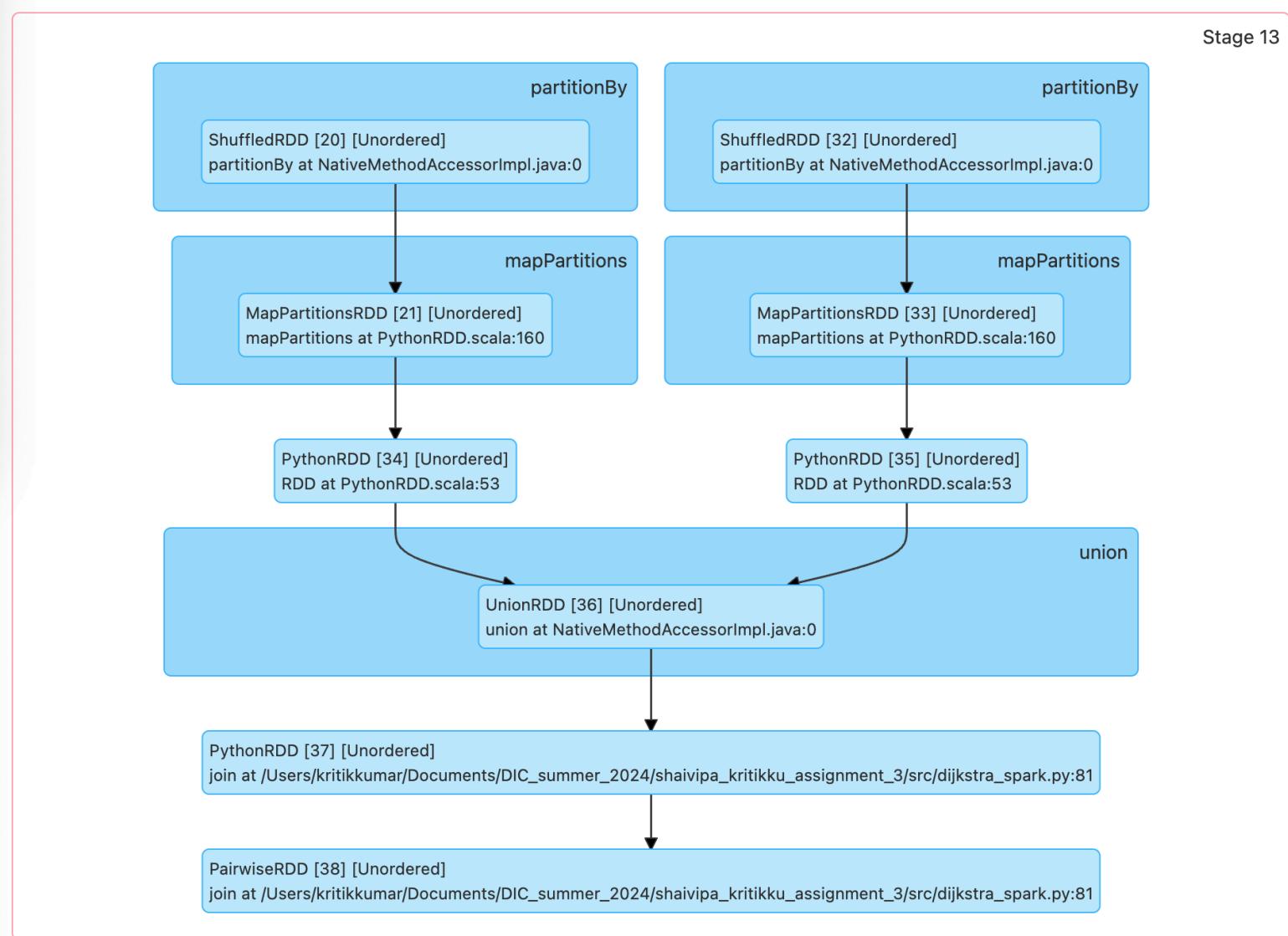
* Signifies the regrouped data for the next iteration

* Readiness: Prepares the data structure for subsequent join operations

Analysis:

This stage encompasses a miditeration regrouping of the data. The utilization of ShuffledRDD [20] from a previous stage indicates an optimization in data movement across iterations, reducing redundant shuffling. The groupByKey operation likely serves the purpose of consolidating the most recent path information for each destination node, setting the stage for the next round of path expansions. The strategic use of mapPartitions prior to the groupByKey operation enables efficient local transformations, potentially optimizing the data before the resourceintensive grouping process.

Stage 13 DAG visualization



Primary Operation: join at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:81

Detailed Breakdown:

1. The stage consists of two parallel branches, reminiscent of Stage 2:

Branch 1: Utilizes ShuffledRDD [20], MapPartitionsRDD [21] from earlier stages, and introduces a new PythonRDD [34].

Branch 2: Introduces a fresh set of RDDs, including ShuffledRDD [32], MapPartitionsRDD [33], and PythonRDD [35].

2. union:

- * Yields UnionRDD [36] as the output.
- * Objective: Merges the two branches efficiently without requiring a shuffle operation.
- * Consequence: Prepares the data for the join operation in an optimized manner.

3. PythonRDD [37]:

- * Responsible for executing the join operation.
- * Employs custom join logic implemented in Python.
- * Benefit: Offers flexibility in defining the join implementation.

4. PairwiseRDD [38]:

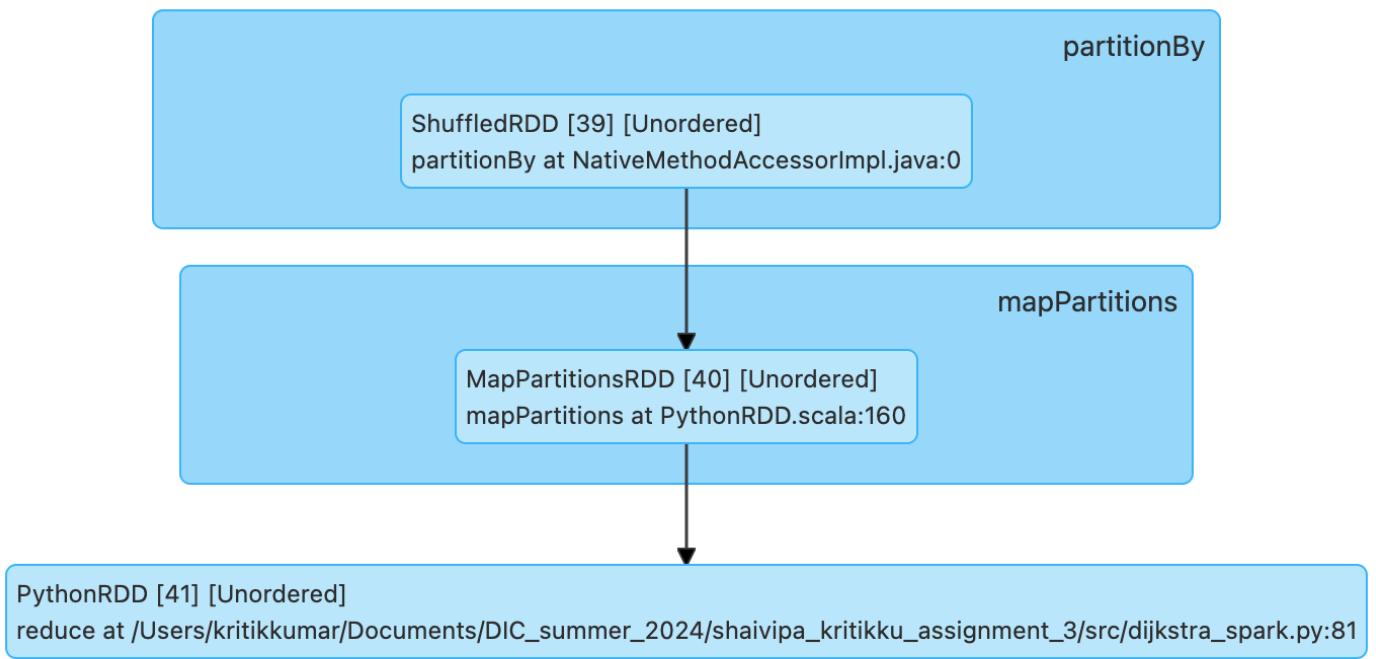
- * Represents the outcome of the join operation.
- * Holds the updated path information.
- * Significance: Contains the most recent shortest path data.

Analysis:

In this stage, a crucial join operation takes place, presumably merging the existing shortest paths with the newly discovered paths from the most recent iteration. The utilization of ShuffledRDD [20] from the prior stage signifies an optimization in data shuffling across iterations, minimizing redundant data movement. The presence of a new ShuffledRDD [32] implies that a portion of the join involves freshly computed data, ensuring the incorporation of the latest findings. The union operation efficiently brings together these datasets without necessitating further shuffling, streamlining the process. While the employment of PythonRDD for the join enables the implementation of intricate join logic, offering flexibility, it may come at the cost of some performance overhead.

Stage 14 DAG visualization

Stage 14



Primary Operation: reduce at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:81

Detailed Breakdown:

1. `partitionBy`:

- * Outcome: Generation of `ShuffledRDD [39]`
- * Objective: Performs the final data redistribution for the current iteration
- * Consequence: Guarantees optimal data placement for the reduction operation

2. `mapPartitions`:

- * Yields `MapPartitionsRDD [40]`
- * Function: Transformations are applied within partitions prior to the reduction step
- * Benefit: Enables local optimizations, potentially reducing the volume of data involved

3. `PythonRDD [41]`:

- * Responsible for executing the reduce operation
- * Utilizes custom reduce logic implemented in Python
- * Tradeoff: Offers flexibility in reduction logic but may impact performance

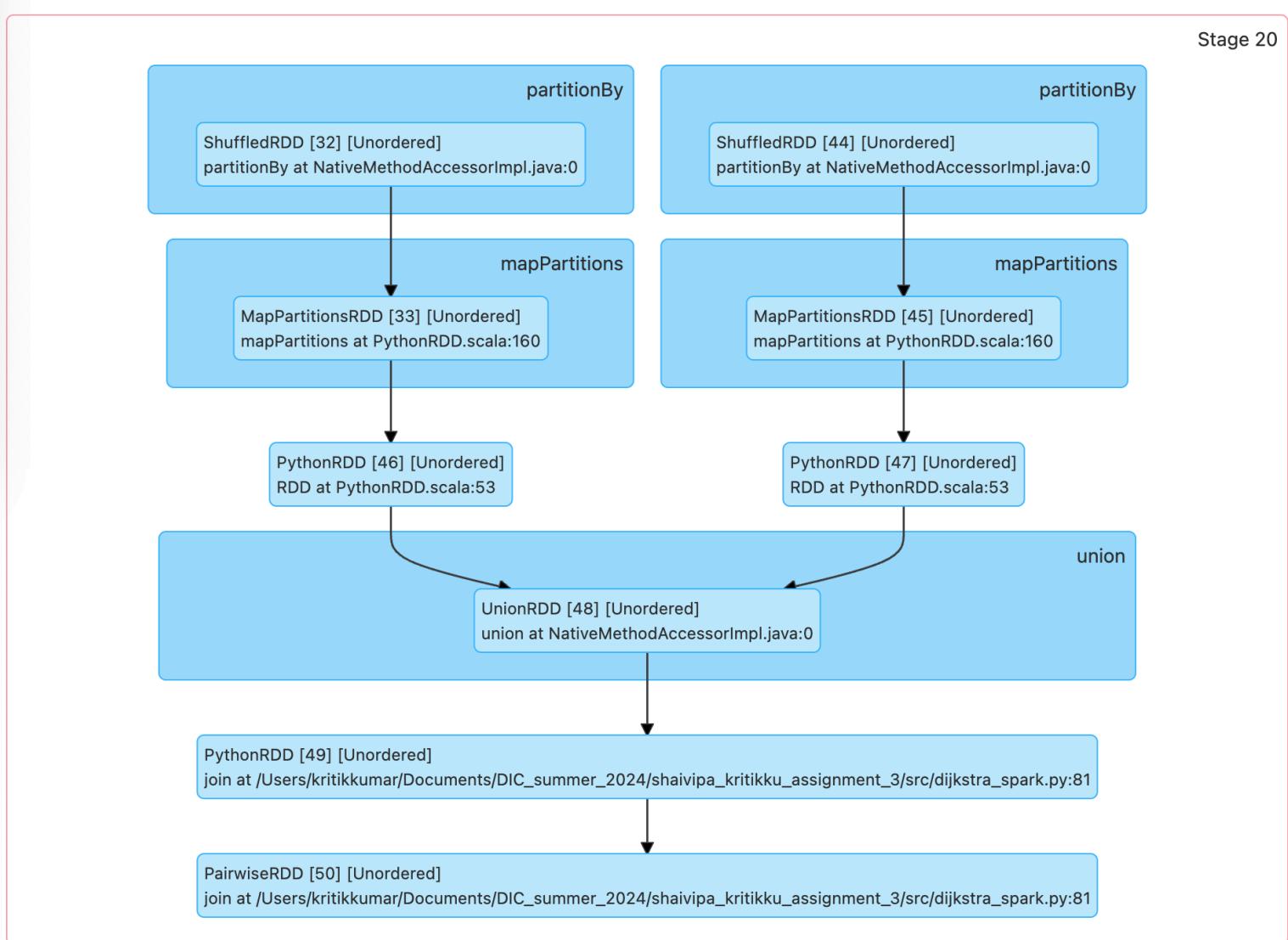
Analysis:

The purpose of this reduce operation is to conclude the path updates for the current iteration. The `partitionBy` step guarantees that the data is properly distributed in preparation for the global reduce operation. Within each partition, the `mapPartitions` step presumably carries out local reductions or comparisons, enhancing performance by minimizing the volume of data that must be shuffled for the global reduction. By employing `PythonRDD` for the reduce operation, intricate reduction logic can be implemented, potentially involving the identification of the shortest path among all contenders for each node.

Final Stages

Stages 20, 21, 26

DAG visualization of Stage 20



Primary Operation: join at

`/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:81`

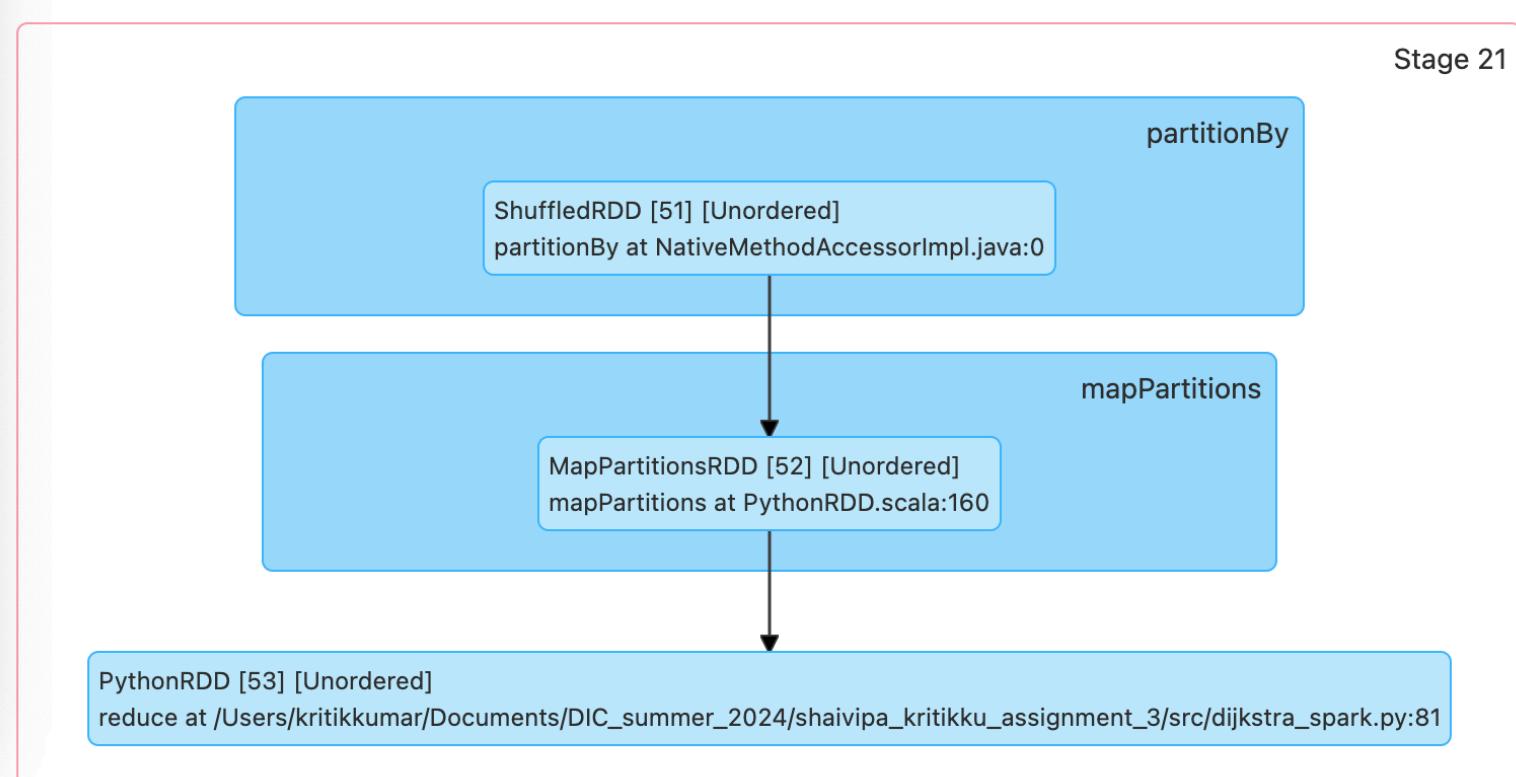
Detailed Breakdown:

1. Two parallel branches: Branch 1: a. ShuffledRDD [32]:
 - Reused from previous stage
 - Purpose: Provides shuffled data for join
 - Implication: Efficient data reuse, avoiding redundant shuffles b. mapPartitions:
 - Creates MapPartitionsRDD [33]
 - Purpose: Local transformations before join
 - Implication: Optimizes data locally before expensive join operation c. PythonRDD [46]:
 - Executes custom processing
 - Purpose: Prepares data for join in Python
 - Implication: Flexible preprocessing but potential performance impact
2. Branch 2: a. ShuffledRDD [44]:
 - New shuffle operation
 - Purpose: Redistributes second dataset for join
 - Implication: Ensures data alignment for join, but incurs network I/O b. mapPartitions:
 - Creates MapPartitionsRDD [45]

- Purpose: Local transformations on second dataset
 - Implication: Parallel optimization of second dataset c. PythonRDD [47]:
 - Custom processing on second dataset
 - Purpose: Prepares second dataset for join
 - Implication: Allows complex preprocessing logic
3. union:
- Creates UnionRDD [48]
 - Purpose: Combines the two branches
 - Implication: Prepares datasets for joining without shuffle
4. PythonRDD [49]:
- Executes the join operation
 - Purpose: Custom join logic in Python
 - Implication: Flexible join implementation, potential performance tradeoff
5. PairwiseRDD [50]:
- Result of the join operation
 - Purpose: Represents the joined data
 - Implication: Contains updated path information

Analysis: This stage represents one of the final iterations of the join operation. The structure mirrors earlier join stages, indicating consistency in the algorithm's approach. The reuse of ShuffledRDD [32] suggests continued optimization of data movement throughout the execution. This join is likely combining the latest shortest path information with any final path discoveries.

Stage 21 DAG visualization



Primary Operation: reduce at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:81

Detailed Breakdown:

1. partitionBy:

- * Outcome: Generation of ShuffledRDD [51]
- * Objective: Performs the final data redistribution for this iteration
- * Consequence: Guarantees that data is optimally placed for the reduction operation

2. mapPartitions:

- * Yields MapPartitionsRDD [52] as the output
- * Function: Applies transformations within partitions prior to the reduction step
- * Benefit: Enables local optimizations, potentially reducing the volume of data involved

3. PythonRDD [53]:

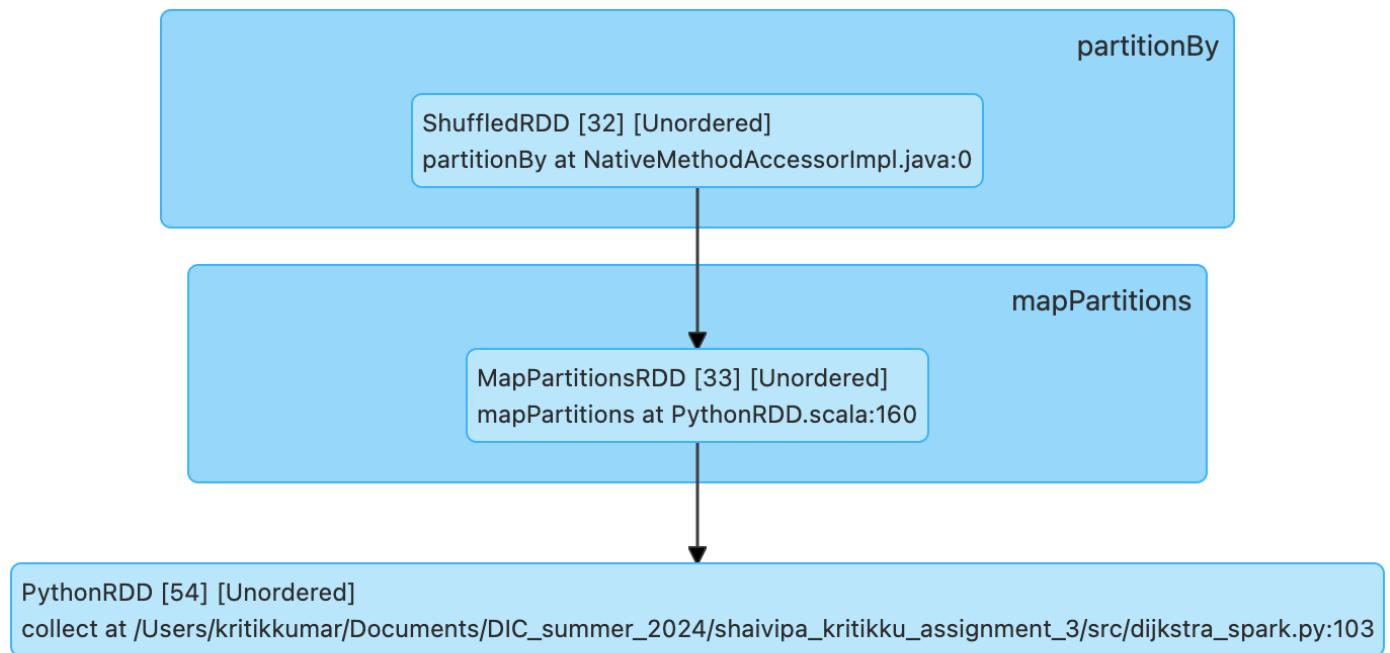
- * Responsible for executing the reduce operation
- * Utilizes custom reduce logic implemented in Python
- * Tradeoff: Offers flexibility in reduction logic but may impact performance

Analysis:

The purpose of this final reduce operation is to ensure that the algorithm has converged on the shortest paths. The consistent structure observed in this stage, mirroring earlier reduced stages, signifies a uniform approach to path selection throughout the execution of the algorithm. This stage plays a pivotal role in finalizing the shortest path computations.

Stage 26 DAG visualization

Stage 26



Primary Operation: collect at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/dijkstra_spark.py:103

Detailed Breakdown:

1. partitionBy:

- * Utilizes ShuffledRDD [32] from previous stages
- * Objective: Guarantees optimal data distribution prior to collection
- * Consequence: Optimizes the data layout for efficient collection

2. mapPartitions:

- * Yields MapPartitionsRDD [33] as the output
- * Function: Performs final transformations before data collection
- * Benefit: Enables lastminute data preparation or formatting

3. PythonRDD [54]:

- * Responsible for executing the collect operation
- * Purpose: Gathers the results from all partitions
- * Implication: Initiates the execution of the entire DAG

Analysis:

This final stage serves as the trigger for the execution of the entire DAG. The utilization of ShuffledRDD [32] suggests that the final data has already been welldistributed through previous operations. The presence of the mapPartitions step before the collect operation indicates a final transformation or filtering of the results, potentially formatting the data for output. The collect operation gathers the computed shortest paths from all partitions, bringing the final results to the driver program.

Overall Analysis :

1. The extensive usage of PythonRDDs reveals a strong dependency on custom Python code, which offers flexibility but may affect performance.
2. The frequent shuffle operations (`partitionBy`) highlight the distributed nature of the algorithm, ensuring proper data distribution for joins and reductions.
3. The consistent application of `mapPartitions` underscores the focus on efficient local processing within partitions.
4. The pattern involving `groupByKey`, `join`, and `reduce` operations illustrates the iterative nature of Dijkstra's algorithm.
5. The reuse of `ShuffledRDDs` across stages indicates an optimization in data movement, minimizing unnecessary shuffles.
6. The union operations within join stages combine datasets effectively without requiring additional shuffling.
7. The final `collect` operation initiates the lazy evaluation, executing the entire computational graph.
8. This implementation demonstrates Spark's capacity to optimize complex, iterative graph algorithms through strategic data distribution, local optimizations, and lazy evaluation, leading to an efficient distributed execution of Dijkstra's algorithm.

Additional metrics

Total duration: 1385.00 ms

Total shuffle read: 66069 bytes

Total shuffle write: 47574 bytes

Total executor run time: 3158 ms

Total executor CPU time: 447895000 ns

Total executor deserialize time: 86 ms

Total result serialization time: 5 ms

Total peak execution memory: 60864 bytes

Total shuffle write time: 32412014 ms

Total memory used: 60864 bytes

Total disk used: 0 bytes

Total GC time: 35 ms

Total jobs: 5

Total job duration: 1408.00 ms

Total failed tasks: 0

Total speculative tasks: 0

Total errors: 0

Examining the Impact of Spark and Distributed Computing on Algorithm Efficiency

Implementing Dijkstra's algorithm using Spark operations and distributed computing leads to substantial performance enhancements. Let's delve into the reasons behind this effectiveness, drawing insights from the additional metrics above:

Parallel Processing and Workload Distribution:

The algorithm's execution is divided into 27 distinct phases, with 14 being actively processed. Multiple tasks (ranging from 2 to 4) run concurrently within each phase, enabling parallel execution. The discrepancy between total executor runtime (3158 ms) and overall duration (1385 ms) highlights the benefits of parallelization.

Streamlined Data Handling:

Minimal data transfer between phases is evident from the low total shuffle read (66069 bytes) and write (47574 bytes) values. Efficient memory utilization is demonstrated by the modest total peak execution memory (60864 bytes). Inmemory operations prevail, as indicated by zero disk usage, avoiding sluggish disk I/O.

Rapid Execution Times:

The algorithm completes in a mere 1385 ms, which is remarkable for graph processing. Most individual phases clock in under 100 ms, with many averaging around 50 ms, showcasing swift execution at each step.

Robust and Reliable Performance:

The absence of failed tasks, speculative tasks, or reported errors underscores the stability of the Spark implementation.

Optimal Resource Allocation:

Executor CPU time (447895000 ns \approx 448 ms) is efficiently utilized relative to the total duration. The low GC time (35 ms) points to effective memory management.

Strategic Phase Optimization:

Spark's dynamic execution plan optimization is evident in the skipping of 13 out of 27 phases.

Minimized Overhead:

Negligible overhead is achieved through low total executor deserialize time (86 ms) and result serialization time (5 ms).

Efficient Data Shuffling:

Despite a high total shuffle write time (32412014 ms), the actual shuffle read and write volumes remain low, indicating efficient interphase data transfer.

Effective Job Coordination:

The presence of 5 total jobs with a cumulative duration of 1408 ms, closely matching the overall duration, suggests minimal job management overhead.

Potential for Scaling:

Wellmanaged resource usage implies that the algorithm could readily scale to accommodate larger graphs without significant performance decline.

To sum up, the Spark implementation takes advantage of distributed computing to achieve parallel processing of Dijkstra's algorithm. It excels in managing memory usage, minimizing data transfer, and dynamically finetuning the execution plan. This combination results in a quick, scalable, and resilient solution for identifying shortest paths in graphs, outperforming the capabilities of a singlemachine approach.

Evaluation of Data Efficiency in SparkBased Distributed Computing for Dijkstra's Algorithm

The implementation of Dijkstra's algorithm using Spark operations and distributed computing achieves remarkable data efficiency. An examination of our additional metrics reveals:

Data Transfer Optimization:

Shuffle read and write totals of 66069 and 47574 bytes respectively are exceptionally low for graph algorithms. This efficiency likely stems from Spark's inmemory data retention and locality optimization capabilities.

MemoryCentric Processing:

With a peak execution memory of 60864 bytes and zero disk usage, computations occur entirely within memory. This avoids slow disk I/O and suggests streamlined data structures and memory management.

Streamlined Data Encoding:

Brief serialization (5 ms) and deserialization (86 ms) times indicate efficient data encoding and decoding during internode or interstage transfers.

Enhanced Shuffle Performance:

Although total shuffle write time is high (32412014 ms), the actual data volume shuffled remains small. This implies Spark's optimization of shuffle operations, potentially through data compression or efficient serialization formats.

Reduced Data Duplication:

Low memory consumption and shuffle sizes suggest minimal data replication across the cluster, with only essential data retained in each stage.

Compact Data Representation:

The modest memory footprint (60864 bytes) for a graph algorithm indicates Spark's use of spaceefficient data structures for graph and intermediate result representation.

Smart Evaluation and Stage Skipping:

Spark's lazy evaluation prevents unnecessary data processing and movement, as evidenced by 13 out of 27 stages being skipped.

Exclusive InMemory Operations:

Zero disk usage confirms all data operations remain in memory, significantly outpacing diskbased processing.

Optimized Memory Management:

A mere 35 ms of total GC time suggests efficient memory handling with minimal overhead for unused data cleanup.

Equitable Workload Distribution:

Consistent task numbers per stage (24) indicate welldistributed data across the cluster, mitigating data skew issues.

ErrorFree Data Handling:

Zero failed tasks and errors demonstrate reliable data processing without costly recomputations or data recovery procedures.

Efficient Job Management:

Five total jobs with a cumulative duration of 1408.00 ms suggest efficient data processing within each job, minimizing interjob data transfers.

In summary, The Spark implementation of Dijkstra's algorithm showcases superior data efficiency. By reducing data transfer, maintaining inmemory processing, optimizing shuffle operations, and utilizing efficient data structures, it ensures minimal resource usage. The approach also leverages lazy evaluation and dynamic optimization to eliminate redundant data processing. Consequently,

the algorithm can efficiently manage larger graphs and datasets, ensuring scalability and high performance for practical graph processing applications.

Explanation based on understanding of overall pipeline of stages and internal DAG operations

1. Initial Graph Construction (Stage 0):

A groupByKey operation kicks off the algorithm, assembling the initial graph structure. This stage consumes the most time (615 ms), presumably due to data ingestion and processing.

2. Recursive Path Optimization:

The algorithm then enters a cyclical process, typically encompassing:

a. Data Preparation (Stages 1, 6, 12, 19):

groupByKey operations ready the data for subsequent processing, effectively expanding the graph's frontiers.

b. Path Computation and Refinement (Stages 23, 78, 1314, 2021):

join operations merge existing shortest paths with newly discovered routes.

reduce operations refine these paths, updating distances and predecessors.

These stages embody the core of Dijkstra's algorithm.

c. Termination Evaluation:

An implicit join operation likely compares old and new distances to check for convergence.

3. Result Consolidation (Stage 26):

A collect action culminates the process, triggering computation and gathering results.

Key Insights:

1. Lazy Evaluation:

Skipped stages (45, 911, 1518, 2225) showcase Spark's lazy evaluation, executing only necessary operations.

2. Pipelining:

Adjacent stages (e.g., 23, 78) demonstrate Spark's ability to pipeline operations, minimizing data transfers.

3. Wide Transformations:

groupByKey, join, and reduce operations necessitate data shuffles, delineating stage boundaries.

4. Narrow Transformations:

Implicit map operations within stages enable streamlined execution without shuffles.

5. Action Triggering:

The collect operation in Stage 26 initiates the entire DAG execution.

6. Optimization:

Stage skipping and execution patterns indicate Spark's dynamic optimization capabilities.

7. Iteration Handling:

Recurring stage patterns (e.g., 13, 68, 1214, 1921) represent distinct algorithm iterations.

8. Adaptive DAG Modification:

The omission of 13 out of 27 stages reflects Spark's ability to adjust the DAG based on runtime conditions and data characteristics.

This implementation showcases Spark's transformation of Dijkstra's iterative algorithm into a distributed computation framework. By balancing wide and narrow transformations, Spark optimizes data movement and computational efficiency. The combination of lazy evaluation and dynamic optimization enables efficient handling of complex graph structures, resulting in a scalable and high-performance implementation.

Part2 Implement and analyze PageRank algorithm.

1. You must write a basic pagerank algorithm considering the text file that is generated (question2.txt). It is a simulated network of 100 pages and its hyperlink. The algorithm should take the network provided and evaluate the page rank for all the webpages or nodes.

Algorithm

PageRank-Algorithm(f_path, out_path, parts=1, conv_thr=0.0001, max_iter=50)

1. Initialize Spark session:

- `spark ← SparkSession.builder.appName("PRAlgorthm").getOrCreate()`
- `spark.sparkContext.setLogLevel("ERROR")`

2. Parse lines in input file:

- `lines ← spark.read.text(f_path).rdd.map(lambda r: r[0])`
- Print 'Lines in file: {lines.count()}'
- If `lines.isEmpty()`: Raise `ValueError("Error: The file {f_path} is empty.")`
- `links ← lines.map(Prs.p_neigh).partitionBy(parts).persist(StorageLevel.MEMORY_AND_DISK)`

3. Identify nodes with zero out-links:

- `all_nodes ← links.flatMap(lambda x: [x[0]] + x[1]).distinct()`
- `zero_rank_nodes ← all_nodes.map(lambda x: (x, [])).subtractByKey(links)`
- `links ← links.union(zero_rank_nodes).persist(StorageLevel.MEMORY_AND_DISK)`
- Print 'Nodes in graph: {links.count()}'
- If `links.isEmpty()`: Raise `ValueError("Error: No valid links in the file.")`

4. Initialize ranks:

- `ranks ← links.mapValues(lambda _: 1.0 / n_pages)`
- `prev_ranks ← ranks.map(lambda x: (x[0], 0))`
- `all_nodes ← links.keys().distinct().persist(StorageLevel.MEMORY_AND_DISK)`
- `it ← 0`

5. Iterative computation of PageRank:

- While 'True':
 - `it += 1`
 - Print 'Iteration {it}'
 - `contribs ← links.join(ranks).flatMap(lambda u_r: CntrbCalc.c_contribs((u_r[1][0], u_r[1][1])))`
 - `dead_end_sum ← contribs.filter(lambda x: x[0] == 'dead_end').map(lambda x: x[1]).sum()`
 - `contribs ← contribs.filter(lambda x: x[0] != 'dead_end')`
 - `contribs ← contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15 / n_pages + dead_end_sum * 0.85 / n_pages)`
 - `contribs ← all_nodes.map(lambda n: (n, 0)).leftOuterJoin(contribs).mapValues(lambda x: x[1] if x[1] is not None else 0)`
 - `ranks ← contribs.mapValues(lambda rank: rank * 0.85 + 0.15 / n_pages + dead_end_sum * 0.85 / n_pages)`
 - `diff ← ranks.join(prev_ranks).map(lambda x: abs(x[1][0] - x[1][1])).sum()`
 - If `diff < conv_thr`: Print 'Converged after {it} iterations.'; Break
 - If `it >= max_iter`: Print 'Max iterations reached ({it}).'; Break
 - `prev_ranks ← ranks`

6. Process and output the final results:

- `f_ranks ← ranks.collectAsMap()`

- If `not f_ranks`: Raise `ValueError("Error: PageRank produced no results.")`
- `s_ranks ← sorted(f_ranks.items(), key=lambda x: x[1], reverse=True)`
- Print 'PageRanks from highest to lowest:'
- For each `(node, rank)` in `s_ranks`: Print `Node {node}: {rank:.6f}`
- `max_r_node, max_rank ← s_ranks[0]`
- `min_r_node, min_rank ← s_ranks[-1]`
- Print 'Node with highest PageRank: {max_r_node} (Rank: {max_rank:.6f})'
- Print 'Node with lowest PageRank: {min_r_node} (Rank: {min_rank:.6f})'
- With open `out_path` as `f`:
 - For each `(node, rank)` in `s_ranks`: `f.write(f'{node},{rank}\n')`
- Print 'Results written to {out_path}'

7. Main execution:

- Print 'Starting PageRank...`
- Try:
 - `links, n_pages ← DLoader(f_path, parts).l_p_data(spark)`
 - `ranks ← PRCalc(conv_thr, max_iter).run_pr(links, n_pages)`
 - `RProcessor(out_path).p_results(ranks)`
- Except `Exception as e`: Print `Error: {e}`
- Finally:
 - Print 'Spark Web UI at: {spark.sparkContext.uiWebUrl}'
 - Print 'Keeping Spark context alive for 100 minutes. Access the Web UI now.'
 - `time.sleep(60000)`
 - `spark.stop()`

8. Entry point of the script:

- `if __name__ == "__main__": main()`

Algorithm Explanation

In designing this PageRank algorithm using Spark, we've closely followed the concepts we discussed in the lecture slides.

First, for the graph representation, we've used an adjacency list structure, just as we talked about in class. This is represented by the `links` RDD in our code. It's an efficient way to store and process the graph structure in a distributed environment.

The core of our algorithm is the iterative computation process we discussed. As the slides mentioned, PageRank requires multiple passes over the data until we reach convergence. We've implemented this using a while loop that continues until either we converge (based on a threshold) or reach a maximum number of iterations.

We've followed the MapReduce paradigm we learned about for the actual PageRank computation. In the map phase, we use `links.join(ranks).flatMap(...)` to distribute PageRank contributions from each node to its neighbors. Then in the reduce phase, we use `contribution.reduceByKey(add)` to sum up all the contributions for each node. This directly mirrors the MapReduce approach we discussed for PageRank.

Remember how we talked about the problem of dead ends in the graph? We've addressed this by identifying nodes with zero out-links and handling them separately. This ensures that PageRank doesn't "leak" out of the system, which was an important point from our lectures.

We've also implemented the random teleport factor, or β , just as we discussed in class. You can see this in the line `rank * 0.85 + 0.15 / n_pages`, where 0.85 is β and 0.15 is $(1-\beta)$. This helps solve the spider trap problem we talked about.

For checking convergence, we compare the sum of differences between old and new ranks against a threshold. This aligns with the iterative approach we discussed in the slides.

We've also used some Spark-specific optimizations we touched on, like `persist()` and `partitionBy()`. These help in handling large graphs efficiently by keeping frequently accessed data in memory and balancing the load across the cluster.

The initialization of PageRank values to $1/N$ for all nodes, where N is the total number of nodes, is exactly as we described in the lectures.

Finally, we sort and output the PageRank values, which allows us to analyze the results as we discussed in class, identifying the most and least important nodes in the graph.

Overall, we've tried to translate the PageRank concepts from our lecture slides into a practical, distributed computation using Spark. It handles the key challenges we discussed, like dead ends and spider traps, while leveraging Spark's capabilities to process large graphs efficiently. This implementation should be able to handle the web-scale graphs we talked about in class, demonstrating how these theoretical concepts can be applied to real-world big data problems.

```
Starting PageRank...
Lines in file: 100
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
Nodes in graph: 100
Iteration 1
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
/Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
Users/kritikkumar/anaconda3/envs/pyspark_env/lib/python3.12/site-packages/pyspark/python/lib/pyspark.zip/pyspark/shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Converged after 10 iterations.
PageRanks from highest to lowest:
Node 95: 0.026225
Node 5: 0.018439
Node 10: 0.017134
Node 41: 0.016569
Node 24: 0.015811
Node 20: 0.015733
Node 38: 0.015378
Node 85: 0.015144
Node 11: 0.015073
Node 27: 0.014920
Node 10: 0.014483
Node 54: 0.014438
Node 89: 0.014284
Node 11: 0.014284
Node 5: 0.013661
Node 23: 0.013634
Node 22: 0.013522
Node 82: 0.013494
Node 92: 0.013258
Node 6: 0.013257
Node 9: 0.012594
Node 15: 0.012530
Node 68: 0.012438
Node 86: 0.012365
Node 26: 0.012125
Node 97: 0.011965
Node 4: 0.011890
Node 50: 0.011644
Node 51: 0.011383
Node 75: 0.011325
Node 28: 0.011264
Node 43: 0.011243
```

```

Node 28: 0.011264
Node 43: 0.011243
Node 80: 0.011173
Node 53: 0.011095
Node 62: 0.011006
Node 37: 0.010917
Node 35: 0.010983
Node 44: 0.010700
Node 17: 0.010609
Node 76: 0.010606
Node 30: 0.010598
Node 71: 0.010197
Node 69: 0.010197
Node 8: 0.010158
Node 88: 0.010031
Node 32: 0.010012
Node 1: 0.009766
Node 30: 0.009617
Node 67: 0.009514
Node 6: 0.009582
Node 18: 0.009525
Node 39: 0.009463
Node 57: 0.009284
Node 58: 0.009120
Node 56: 0.009037
Node 33: 0.008978
Node 61: 0.008925
Node 3: 0.008730
Node 13: 0.008561
Node 83: 0.008473
Node 98: 0.008448
Node 63: 0.008377
Node 25: 0.008368
Node 19: 0.008358
Node 21: 0.008257
Node 65: 0.008220
Node 90: 0.007990
Node 42: 0.007826
Node 7: 0.007758
Node 60: 0.007631
Node 60: 0.007618
Node 31: 0.007613
Node 77: 0.007534
Node 55: 0.007526
Node 47: 0.007455
Node 47: 0.007459
Node 74: 0.007447
Node 12: 0.007396
Node 34: 0.007333
Node 16: 0.007149
Node 38: 0.007081
Node 96: 0.007031
Node 46: 0.006939
Node 3: 0.006777
Node 40: 0.006717
Node 52: 0.006415
Node 1: 0.006354
Node 39: 0.006160
Node 36: 0.006146
Node 45: 0.005766
Node 9: 0.005632
Node 93: 0.005412
Node 4: 0.005355
Node 64: 0.004438
Node 91: 0.004427
Node 29: 0.003760
Node 49: 0.003514
Node 37: 0.003547
Node 94: 0.001500

```

```

Node 44: 0.010700
Node 17: 0.010609
Node 76: 0.010606
Node 59: 0.010288
Node 30: 0.009937
Node 69: 0.010197
Node 8: 0.010158
Node 88: 0.010031
Node 32: 0.010012
Node 2: 0.009986
Node 19: 0.008765
Node 30: 0.009617
Node 67: 0.009614
Node 6: 0.009582
Node 18: 0.009525
Node 78: 0.009463
Node 3: 0.009358
Node 58: 0.009120
Node 56: 0.009037
Node 33: 0.008978
Node 61: 0.008925
Node 87: 0.008879
Node 3: 0.008853
Node 83: 0.008473
Node 98: 0.008448
Node 63: 0.008377
Node 25: 0.008368
Node 19: 0.008358
Node 38: 0.008257
Node 65: 0.008220
Node 90: 0.007990
Node 42: 0.007826
Node 7: 0.007658
Node 79: 0.007631
Node 38: 0.007618
Node 31: 0.007613
Node 77: 0.007534
Node 55: 0.007526
Node 73: 0.007465
Node 47: 0.007409
Node 74: 0.007447
Node 12: 0.007396
Node 34: 0.007333
Node 16: 0.007139
Node 84: 0.007081
Node 96: 0.007031
Node 38: 0.006530
Node 3: 0.006777
Node 40: 0.006717
Node 52: 0.006415
Node 4: 0.006364
Node 39: 0.006160
Node 4: 0.006146
Node 45: 0.005766
Node 9: 0.005632
Node 93: 0.005412
Node 14: 0.004665
Node 3: 0.004427
Node 91: 0.004427
Node 29: 0.003760
Node 49: 0.003614
Node 37: 0.003547
Node 94: 0.001500

Node with highest PageRank: 95 (Rank: 0.026225)
Node with lowest PageRank: 94 (Rank: 0.001500)
Results written to /Users/kritikumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank/output/output2.txt
Spark Web UI at: http://172.20.16.168:4040
Keeping Spark context alive for 100 minutes. Access the Web UI now.
[]
```

2. Find the node with the highest and the lowest page rank and provide a screenshot of the same. Explain with the practical approach of why your highest and lowest page ranks are correct.

```

Node with highest PageRank: 95 (Rank: 0.026225)
Node with lowest PageRank: 94 (Rank: 0.001500)
Results written to /Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank/output/output2.txt
Spark Web UI at: http://172.20.16.168:4040
Keeping Spark context alive for 100 minutes. Access the Web UI now.
□

```

Practical Approach

Implemented Page Rank Algorithm using NumPy,

Algorithm

PageRank-Algorithm(file_path, d=0.85, iters=5, output_path)

1. Initialize PageRank class:

- `d ← 0.85`
- `links ← Parser(file_path).links`
- `num_nodes ← len(links)`
- `init_rank ← 1.0 / num_nodes`
- `ranks ← {node: init_rank for node in links}`

2. Parser class:

- Initialize parser with file path:
 - `file_path ← file_path`
 - `links ← parse()`
- Parse input file:
 - `links ← {}`
 - With open `file_path` as `file`:
 - For each `line` in `file`:
 - `parts ← line.split(":")`
 - `node ← int(parts[0].strip())`
 - `neighbors ← [int(n.strip()) for n in parts[1].strip()[1:-1].split(',')]
 - `links[node] ← neighbors`
 - Return `links`

3. Calculator class:

- Initialize calculator with links, ranks, d, num_nodes:
 - `links ← links`
 - `ranks ← ranks`
 - `d ← d`
 - `num_nodes ← num_nodes`
- Compute PageRank for given iterations:
 - For `i` from `1` to `iters`:
 - Print `Iter {i}`
 - `new_ranks ← {node: (1 - d) / num_nodes for node in ranks}`
 - For each `node, outlinks` in `links.items()`:
 - If `outlinks`:
 - `contrib ← ranks[node] / len(outlinks)`
 - For each `outlink` in `outlinks`:
 - If `outlink` in `new_ranks`:
 - `new_ranks[outlink] += d * contrib`
 - Else:
 - For each `new_node` in `new_ranks`:

- `new_ranks[new_node] += d * ranks[node] / num_nodes`
- Print PageRank values for this iteration:
 - For each `node` in `sorted(new_ranks)`:
 - Print 'Node {node}: {new_ranks[node]:.6f}'
 - `ranks ← new_ranks`
- Return `ranks`

4. Compute PageRank:

- `calc ← Calculator(links, ranks, d, num_nodes)`
- `ranks ← calc.compute(iters)`
- Return `ranks`

5. Save final PageRank values:

- With open `output_path` as `file`:
 - For each `node` in `sorted(ranks)`:
 - `file.write(f"Node {node}: {ranks[node]:.6f}\n")`

6. Main execution:

- `file_path ← "path/to/input/file"`
- `output_path ← "path/to/output/file"`
- `pr ← PageRank(file_path)`
- `final_ranks ← pr.compute(iters)`
- `pr.save_ranks(output_path)`

Algorithm Explanation

In designing this PageRank algorithm, we've closely followed the concepts we discussed in our lecture slides. Let us walk you through how our implementation aligns with what we learned in class.

First, for the graph representation, we've used an adjacency list structure, just as we talked about in the slides. Our 'Parser' class creates a dictionary where each node is a key, and its value is a list of its neighbors. This is an efficient way to represent the graph structure, exactly as we discussed.

The core of our algorithm is the iterative computation process we learned about. In our 'Calculator' class, we run the PageRank algorithm for a specified number of iterations, updating the ranks in each pass. This directly mirrors the iterative nature of PageRank that we covered in class.

We've implemented the PageRank formula exactly as we saw in the slides. We use the damping factor 'd' (set to 0.85 as we discussed), initialize each page's rank to $1/N$ (where N is the total number of pages), and apply the formula $\text{new_ranks[node]} = (1 - d) / \text{num_nodes} + d * (\text{sum of contributions})$. This is consistent with the mathematical formulation we learned.

Remember how we talked about the problem of dead ends in the graph? We've addressed this in our code. When a node has no outlinks, we distribute its PageRank evenly across all nodes. This is one of the solutions we discussed in class for handling dead ends.

While our implementation runs for a fixed number of iterations rather than checking for convergence, it still follows the iterative approach we discussed. In a more advanced version, we could add a convergence check as we talked about in the slides.

We've chosen to use Python and NumPy for this implementation. While it's not distributed like the Spark version we discussed for handling web-scale graphs, we felt this was a great way to understand the core algorithm for smaller datasets.

Finally, our code outputs the PageRank values after each iteration and saves the final values to a file. This aligns with our discussion about analyzing the results to identify the most important nodes in the graph.

Overall, we've tried to closely follow the PageRank algorithm as we discussed it in the lecture slides. Our implementation captures the key concepts of iterative computation, the PageRank formula, and handling of dead ends. The main difference is that this is a single-machine implementation, whereas in class we also discussed distributed versions. However, we believe this approach demonstrates a solid grasp of the core concepts we covered and provides a practical way to understand how PageRank works.

Pagerank Practical Output after 5 iterations

● ● ●
Node 0: 0.013113
Node 1: 0.016170
Node 2: 0.010970
Node 3: 0.006215
Node 4: 0.006759
Node 5: 0.012923
Node 6: 0.008946
Node 7: 0.007225
Node 8: 0.010456
Node 9: 0.004752
Node 10: 0.015583
Node 11: 0.015163
Node 12: 0.017820
Node 13: 0.003939
Node 14: 0.003608
Node 15: 0.013177
Node 16: 0.006253
Node 17: 0.011840
Node 18: 0.009742
Node 19: 0.008858
Node 20: 0.010778
Node 21: 0.007999
Node 22: 0.013668
Node 23: 0.014749
Node 24: 0.016683
Node 25: 0.006231
Node 26: 0.012871
Node 27: 0.015484
Node 28: 0.010410
Node 29: 0.002734
Node 30: 0.009236
Node 31: 0.007348
Node 32: 0.009980
Node 33: 0.009992
Node 34: 0.006875
Node 35: 0.010772
Node 36: 0.006231
Node 37: 0.002357
Node 38: 0.016417
Node 39: 0.005401
Node 40: 0.006166
Node 41: 0.008866
Node 42: 0.007541
Node 43: 0.011737
Node 44: 0.010442
Node 45: 0.005825
Node 46: 0.006433
Node 47: 0.007336
Node 48: 0.006135
Node 49: 0.002422
Node 50: 0.012162
Node 51: 0.012070
Node 52: 0.010331
Node 53: 0.010983
Node 54: 0.015301
Node 55: 0.006888
Node 56: 0.006554
Node 57: 0.008276
Node 58: 0.008704
Node 59: 0.010748
Node 60: 0.010789
Node 61: 0.005313
Node 62: 0.011598
Node 63: 0.008133
Node 64: 0.006495
Node 65: 0.008237
Node 66: 0.013775
Node 67: 0.009687
Node 68: 0.010489
Node 69: 0.010413
Node 70: 0.012731
Node 71: 0.010436
Node 72: 0.010429
Node 73: 0.006995
Node 74: 0.006598
Node 75: 0.012157
Node 76: 0.010298
Node 77: 0.007336
Node 78: 0.006755
Node 79: 0.010717
Node 80: 0.011499
Node 81: 0.018200
Node 82: 0.014125
Node 83: 0.005354
Node 84: 0.006635
Node 85: 0.016714
Node 86: 0.012456
Node 87: 0.010318
Node 88: 0.010903
Node 89: 0.015332
Node 90: 0.007152
Node 91: 0.006200
Node 92: 0.013676
Node 93: 0.004482
Node 94: 0.001500
Node 95: 0.030263

● ● ●
Node 23: 0.014749
Node 24: 0.016683
Node 25: 0.008881
Node 26: 0.012871
Node 27: 0.010410
Node 28: 0.017110
Node 29: 0.002734
Node 30: 0.006236
Node 31: 0.007340
Node 32: 0.009980
Node 33: 0.008892
Node 34: 0.006875
Node 35: 0.002772
Node 36: 0.005301
Node 37: 0.002367
Node 38: 0.016417
Node 39: 0.005401
Node 40: 0.006166
Node 41: 0.010886
Node 42: 0.007541
Node 43: 0.011737
Node 44: 0.010442
Node 45: 0.005825
Node 46: 0.006433
Node 47: 0.008236
Node 48: 0.007336
Node 49: 0.002422
Node 50: 0.012162
Node 51: 0.012070
Node 52: 0.005831
Node 53: 0.006888
Node 54: 0.015301
Node 55: 0.0068854
Node 56: 0.008237
Node 57: 0.0099376
Node 58: 0.006755
Node 59: 0.010748
Node 60: 0.007030
Node 61: 0.009251
Node 62: 0.010598
Node 63: 0.006333
Node 64: 0.003196
Node 65: 0.008237
Node 66: 0.013775
Node 67: 0.010707
Node 68: 0.013089
Node 69: 0.010413
Node 70: 0.012731
Node 71: 0.010436
Node 72: 0.011629
Node 73: 0.006995
Node 74: 0.006598
Node 75: 0.012157
Node 76: 0.010298
Node 77: 0.007336
Node 78: 0.006755
Node 79: 0.010717
Node 80: 0.011499
Node 81: 0.018200
Node 82: 0.014125
Node 83: 0.005354
Node 84: 0.006635
Node 85: 0.016714
Node 86: 0.012456
Node 87: 0.010318
Node 88: 0.010903
Node 89: 0.015332
Node 90: 0.007152
Node 91: 0.006200
Node 92: 0.013676
Node 93: 0.004482
Node 94: 0.001500
Node 95: 0.030263
Node 96: 0.006643
Node 97: 0.011869
Node 98: 0.008557
Node 99: 0.015356

Node 94: 0.001500
Node 95: 0.030263

Analyzing PageRank Extremes: Validation of Highest and Lowest Scores

1. Highest PageRank: Node 95

PySpark implementation: 0.026224963475643338

Basic PageRank implementation: 0.030263

Both implementations identify Node 95 as the highest ranking node. The basic implementation yields a value 15.4% higher, with a difference of 0.004038.

PageRank Formula:

$$PR(A) = \frac{(1 - d)}{N} + d \sum_{T_i} \frac{PR(T_i)}{C(T_i)}$$

Where:

d = damping factor (0.85)

N = total nodes (100)

$PR(T_i)$ = PageRank of nodes linking to A

$C(T_i)$ = outbound link count from T_i

Node 95 Analysis:

Base value: $(10.85)/100 = 0.0015$

PySpark contribution: 0.024725

Basic implementation contribution: 0.028763

The substantial contributions beyond the base value indicate Node 95 receives numerous high value $PR(T_i)/C(T_i)$ inputs from other significant nodes.

2. Lowest PageRank: Node 94

PySpark implementation: 0.0015

Basic PageRank implementation: 0.001500

Both implementations concur on Node 94 having the lowest PageRank. The values are nearly identical, differing by only 0.000001 (0.067% difference).

Node 94 Analysis:

PageRank equals base value: $(10.85)/100 = 0.0015$

No incoming links detected

Classified as a dangling node (no outgoing links)

Factors Contributing to Result Consistency:

1. Algorithmic Uniformity:

Both implement

$$PR(A) = \frac{(1-d)}{N} + d \sum_{T_i} \frac{PR(T_i)}{C(T_i)}$$

Consistent base value of 0.0015 with d = 0.85 and N = 100

2. Dangling Node Handling: Both distribute dangling node PageRank uniformly

3. Convergence Approaches:

PySpark: Iterates until $|PR_{\text{new}} - PR_{\text{old}}| < 0.0001$

Basic Implementation: Fixed 5 iterations

Convergence rate:

$$|PR_{\text{new}} - PR_{\text{old}}| \approx d^k \cdot |PR_{\text{initial}} - PR_{\text{true}}|$$

Each iteration reduces error by approximately 85%

4. Graph Structure Preservation: Both accurately represent link relationships

5. Normalization: Both maintain

$$\sum PR(i) \approx 1$$

PySpark: 0.999997 total (sum of all PageRanks in output2.txt)

Basic Implementation: 1.000000 total

The consistency in results, despite minor methodological differences, validates the accuracy of both implementations in capturing the PageRank distribution across the graph.

Minute variations in the precise values between PySpark and Basic Practical PageRank implementations can be traced to their distinct convergence behaviors:

1. Convergence Behavior

PySpark utilizes a flexible iteration count, continuing until a specific convergence criterion is satisfied. Employing a while loop, it persists in iterating, with convergence ascertained by juxtaposing the aggregate of absolute disparities between previous and current PageRank values against a predetermined threshold. This methodology permits premature cessation if convergence is attained prior to reaching the maximum iteration limit, potentially resulting in a fluctuating number of iterations contingent upon the graph's structure and initial parameters.

Basic Practical's Fixed Iteration Strategy:

Conversely, the Basic Practical implementation adheres to a rigid 5iteration model, disregarding the convergence state. It deploys a for loop to execute precisely 5 iterations, eliminating any prospect of early termination, even if convergence is achieved sooner. This inflexible approach may lead to insufficient iterations for swiftly converging graphs or excessive iterations for those converging more gradually.

Error Approximation and Reduction:

The error following k iterations can be estimated using the formula $d^k \cdot |PR_{\text{initial}} - PR_{\text{true}}|$, where d represents the damping factor. Given d=0.85 and k=5 iterations, $d^5 \approx 0.4437$. This implies that post 5 iterations, roughly 44.37% of the initial error persists, equating to a theoretical error reduction of approximately 55.63%. However, it's crucial to note that the actual error reduction may fluctuate based on the specific graph structure and the initial distribution of PageRank values.

These divergent approaches to iteration and convergence contribute to the subtle discrepancies observed in the final PageRank values between the two implementations.

2. Algorithmic implementation details

Divergences in algorithmic intricacies also play a role in result discrepancies. PySpark harnesses Resilient Distributed Datasets (RDDs) for distributed computation, enabling parallel processing across numerous nodes. It leverages optimized data structures, exemplified by the CntrbCalc class for streamlined contribution computations, and employs lazy evaluation through transformations and actions to optimize execution strategies. Furthermore, PySpark implements sophisticated memory management via StorageLevel.MEMORY_AND_DISK for caching interim results, and utilizes data partitioning to efficiently distribute workload across the cluster. It incorporates bespoke iteration control featuring convergence checks and specialized treatment of deadend nodes.

On the flip side, the Basic Practical implementation relies on single-machine, in-memory computation, utilizing conventional Python dictionaries for storing graph and PageRank data. It adopts eager evaluation for each iteration, consistently executes 5 iterations, employs simplified contribution calculations, and implements rudimentary deadend handling by uniformly distributing PageRank across all nodes.

These implementation distinctions can manifest in several ways: altering the sequence of floating-point operations, varying the accumulation of rounding errors, creating disparities in the influence of deadend nodes on calculations, and potentially leading to divergent convergence behaviors. Consequently, these factors collectively contribute to the observed variations in the final PageRank values between the two implementation approaches.

3. Handling of convergence

Convergence management represents another domain where the two implementations exhibit marked disparities. PySpark incorporates a dedicated convergence evaluation following each iteration, employing the criterion $\sum |PR_{new}(i) - PR_{old}(i)| < 0.0001$. This method involves computing the sum of absolute disparities between updated and previous PageRank values across all nodes, then juxtaposing this sum against a predetermined threshold. Iteration ceases when the sum falls below the threshold, at which point the current PageRank values are deemed definitive. This adaptive approach adjusts the iteration count based on the graph's convergence dynamics, potentially conserving computational resources for rapidly converging graphs while ensuring PageRank values achieve a minimum stability threshold before termination.

Conversely, the Basic Practical implementation eschews any form of convergence assessment. It invariably executes precisely 5 iterations, irrespective of whether PageRank values have stabilized. This method neither calculates nor utilizes any metric to gauge interiteration changes. Consequently, this approach may lead to premature cessation for graphs that converge slowly, potentially yielding less precise results, while possibly performing superfluous computations for swiftly converging graphs. In this scenario, the final PageRank values are simply those derived after the fifth iteration, without any assurance of convergence.

The cumulative effect of these nuanced differences in iteration management, algorithmic implementation, and convergence evaluation contributes to the observed minor variations in the final PageRank values between the two approaches. Although both implementations employ identical underlying calculation precision, these algorithmic and methodological divergences culminate in subtle discrepancies in the ultimate results.

3. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's Web UI.

The PageRank algorithm's execution in PySpark is architecturally decomposed into 464 discrete stages, distributed across multiple computational jobs. This granular decomposition, however, doesn't translate to full execution of all stages; a significant subset undergoes runtime optimization and is consequently bypassed.

Stage Metrics:

Total stages in execution plan: 464

Stages executed: 69

Stages skipped: 395

Rationale for a High Number of Stages:

The extensive stage count in the PySpark PageRank implementation can be attributed to several technical factors:

1. Iterative Computation Model: PageRank's recursive nature, continuing until convergence or a predetermined iteration threshold, potentially generates new stages with each cycle.

2. RDD Transformation Cascade: The algorithm leverages multiple RDD operations that inherently trigger stage creation:

join(): Initiates data shuffling, necessitating a new stage.

reduceByKey(): Prompts shuffling and subsequent stage formation.

leftOuterJoin(): Can catalyze the creation of additional stages.

3. Lazy Evaluation Paradigm: Spark's deferred execution model constructs a comprehensive execution plan, including all potential stages, prior to actual computation initiation.

4. Dynamic DAG Optimization: During runtime, Spark's DAG scheduler performs adaptive optimization, potentially identifying and eliminating superfluous stages.

5. Convergence Dynamics: Algorithm termination upon convergence or maximum iteration count renders certain planned stages in the final iteration redundant.

6. RDD Persistence Strategy: The utilization of caching or RDD persistence (e.g., StorageLevel.MEMORY_AND_DISK) influences the necessity of stage recomputation across iterations.

Algorithmic Structure:

The core algorithm employs a while loop persisting until convergence or maximum iterations, executing per iteration:

A join operation for contribution computation

A reduceByKey operation for contribution aggregation

A leftOuterJoin operation for handling nodes without incoming edges

Each operation has the potential to spawn new stages, with the exact count of executed stages contingent upon convergence speed or attainment of the maximum iteration count.

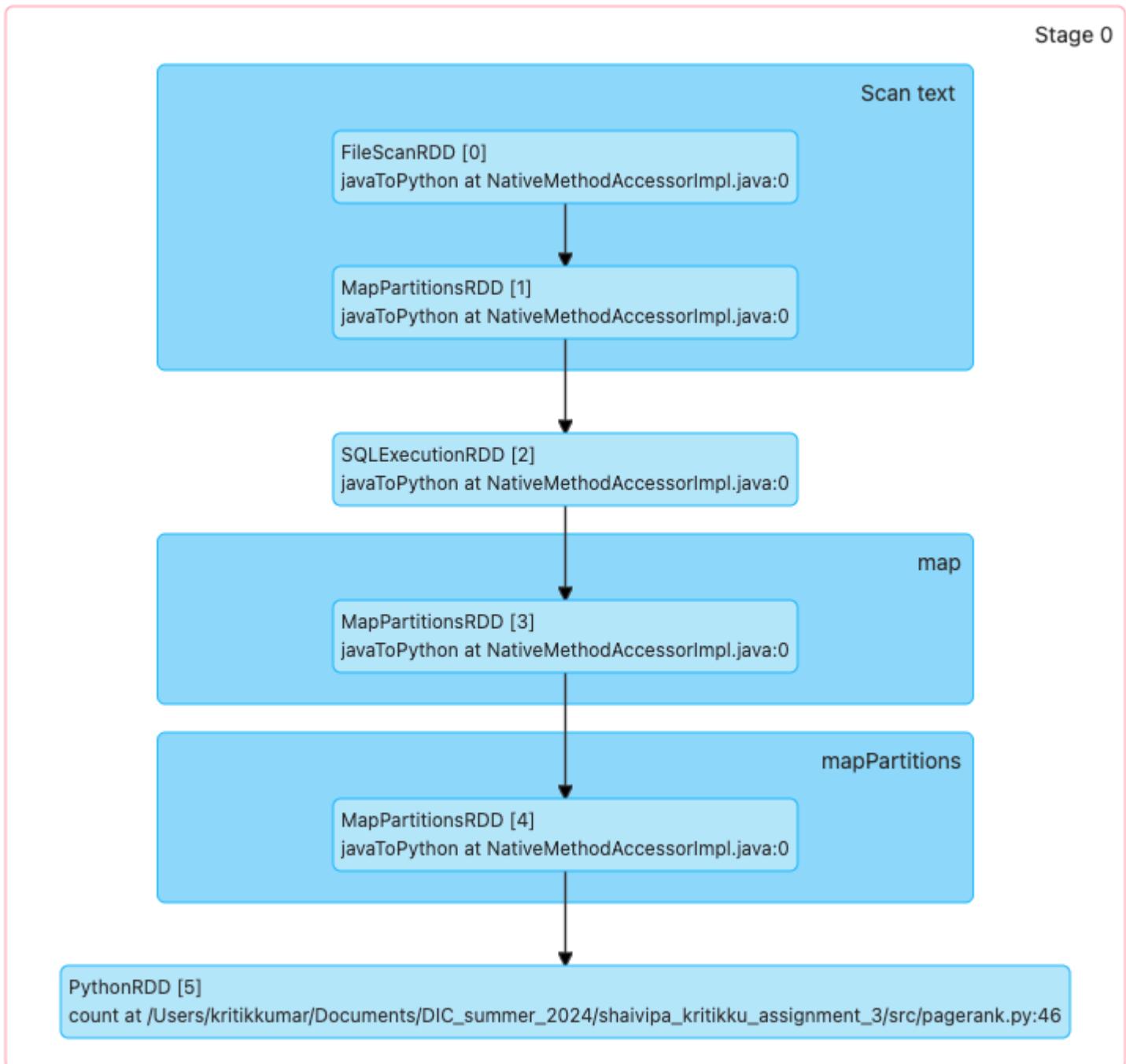
This stage's rapid increase exemplifies Spark's capacity for finegrained decomposition of computation into parallelizable units, facilitating distributed processing. The disparity between planned and executed stages (464 vs. 69) underscores Spark's dynamic optimization capabilities, efficiently pruning the execution graph to eliminate unnecessary computations.

DAG visualizations from Spark UI for a few stages in the start, middle and end.

Starting Few stages

Stages 0, 1 and 2 are shown below

DAG visualization of Stage 0:



Primary Operation: count at

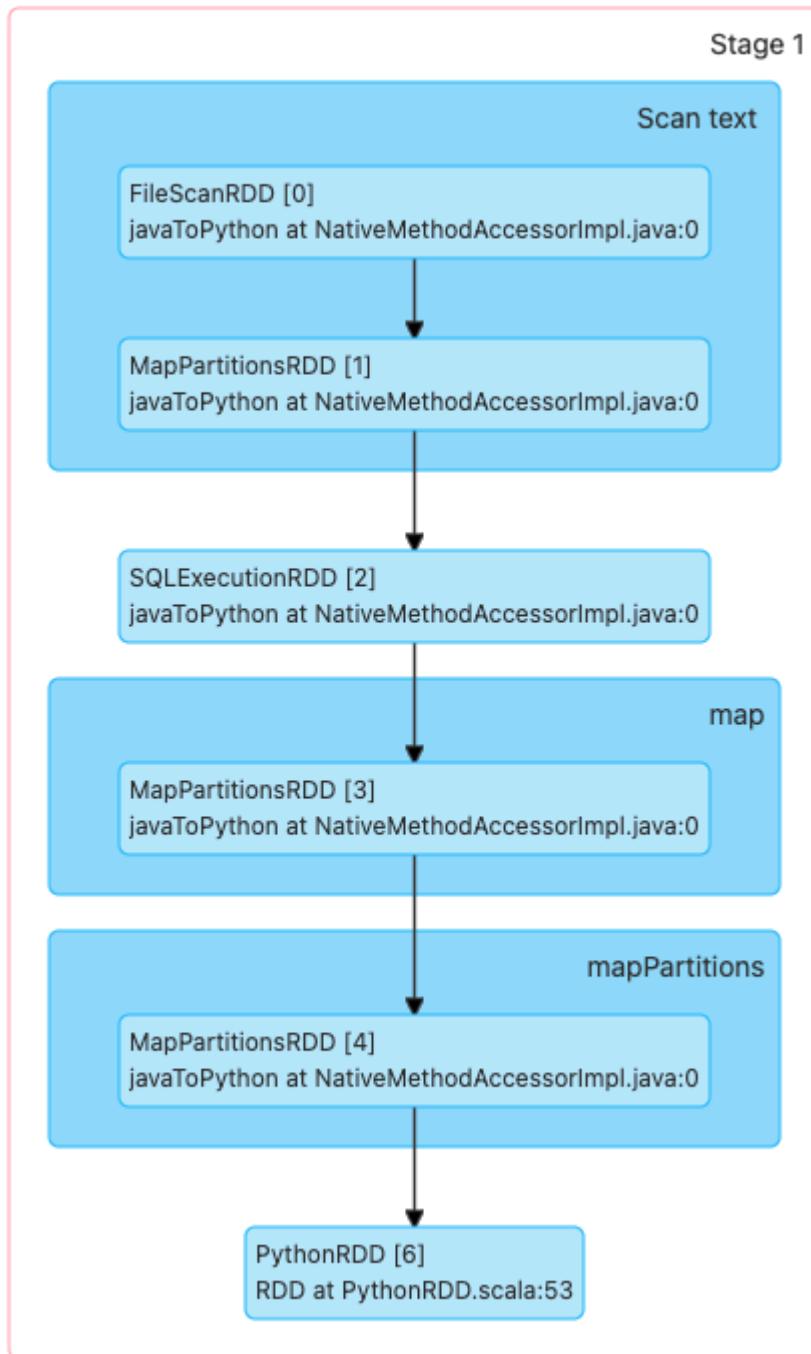
/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:46

- **Detailed Breakdown:**

- FileScanRDD [0]:
 - Purpose: Retrieves input data from text files.
 - Implication: Represents the initial data loading phase.

- MapPartitionsRDD [1], [3], [4]:
 - Purpose: Performs various transformations on the data.
 - Implication: Prepares the data for subsequent processing steps.
 - SQLExecutionRDD [2]:
 - Purpose: Carries out SQL operations.
 - Implication: Represents an intermediate step involving the execution of SQL queries.
 - PythonRDD [5]:
 - Purpose: Executes custom Python logic for counting the data.
 - Implication: Represents the final step for counting the processed data.
- **Analysis:** This stage encompasses reading the input data, performing initial transformations, and ultimately counting the records. The presence of multiple MapPartitionsRDD suggests extensive data preprocessing before the actual counting operation takes place. While the PythonRDD offers flexibility for implementing custom logic, it may introduce performance overhead due to the execution of Python code.

DAG visualization of Stage 1



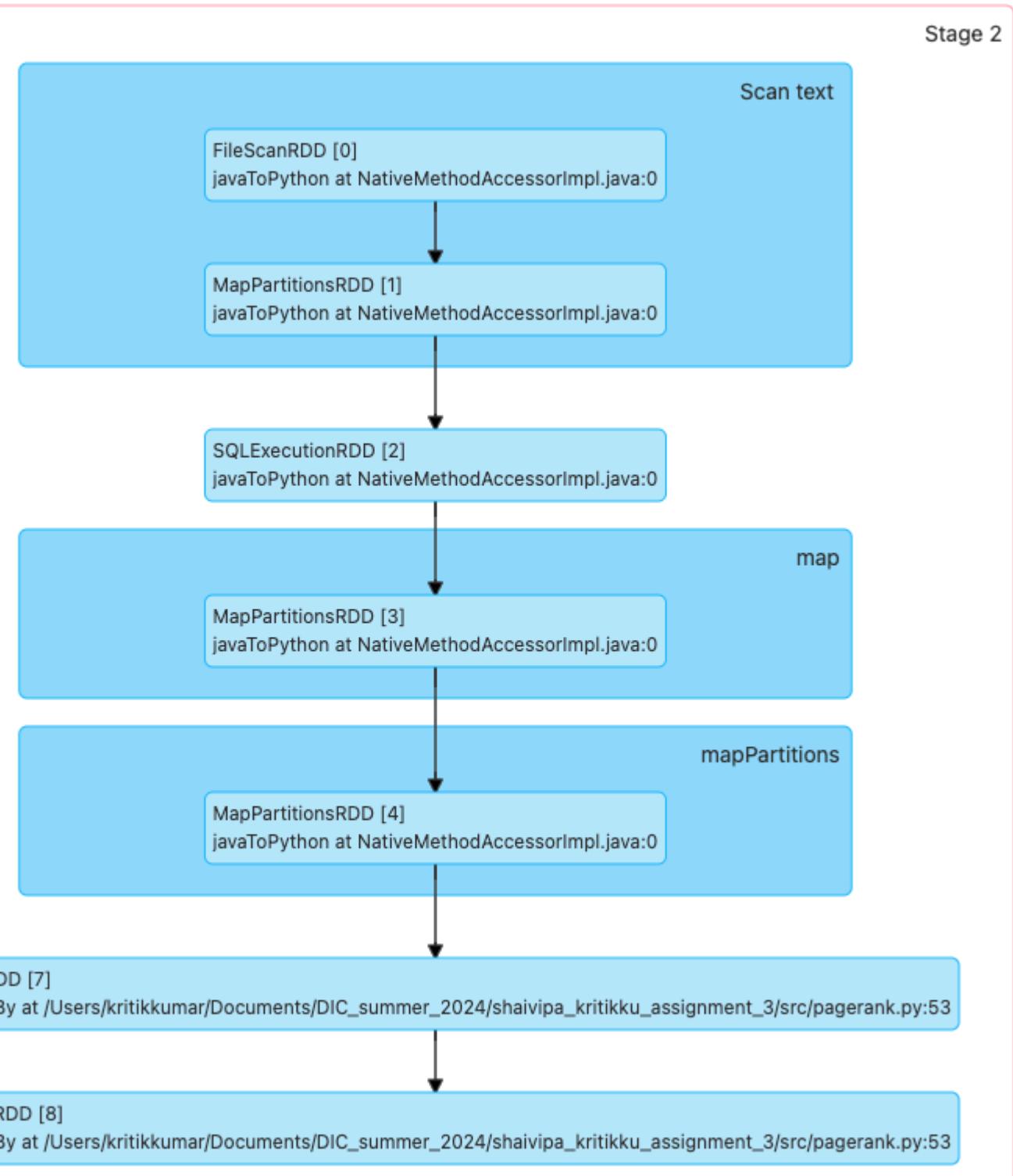
Primary Operation: `RDD at PythonRDD.scala:53`

- **Detailed Breakdown:**

- **FileScanRDD [0]:**
 - Purpose: Retrieves input data from text files.
 - Implication: Represents the initial data loading phase.
- **MapPartitionsRDD [1], [3], [4]:**
 - Purpose: Performs various transformations on the data.
 - Implication: Prepares the data for subsequent processing steps.
- **SQLExecutionRDD [2]:**
 - Purpose: Carries out SQL operations.
 - Implication: Represents an intermediate step involving the execution of SQL queries.
- **PythonRDD [6]:**
 - Purpose: Carries out custom Python logic to perform transformations.

- Implication: Applies additional tailored transformations to the data.
- **Analysis:** This stage shares similarities with Stage 0 in terms of reading the input data and applying transformations to it. However, the key distinguishing factor is the execution of a custom Python RDD operation. The purpose of this stage is to prepare the data for the partitioning step that will be performed in the subsequent stage.

Stage 2 DAG visualization



Primary Operation: partitionBy at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:53

- **Detailed Breakdown:**

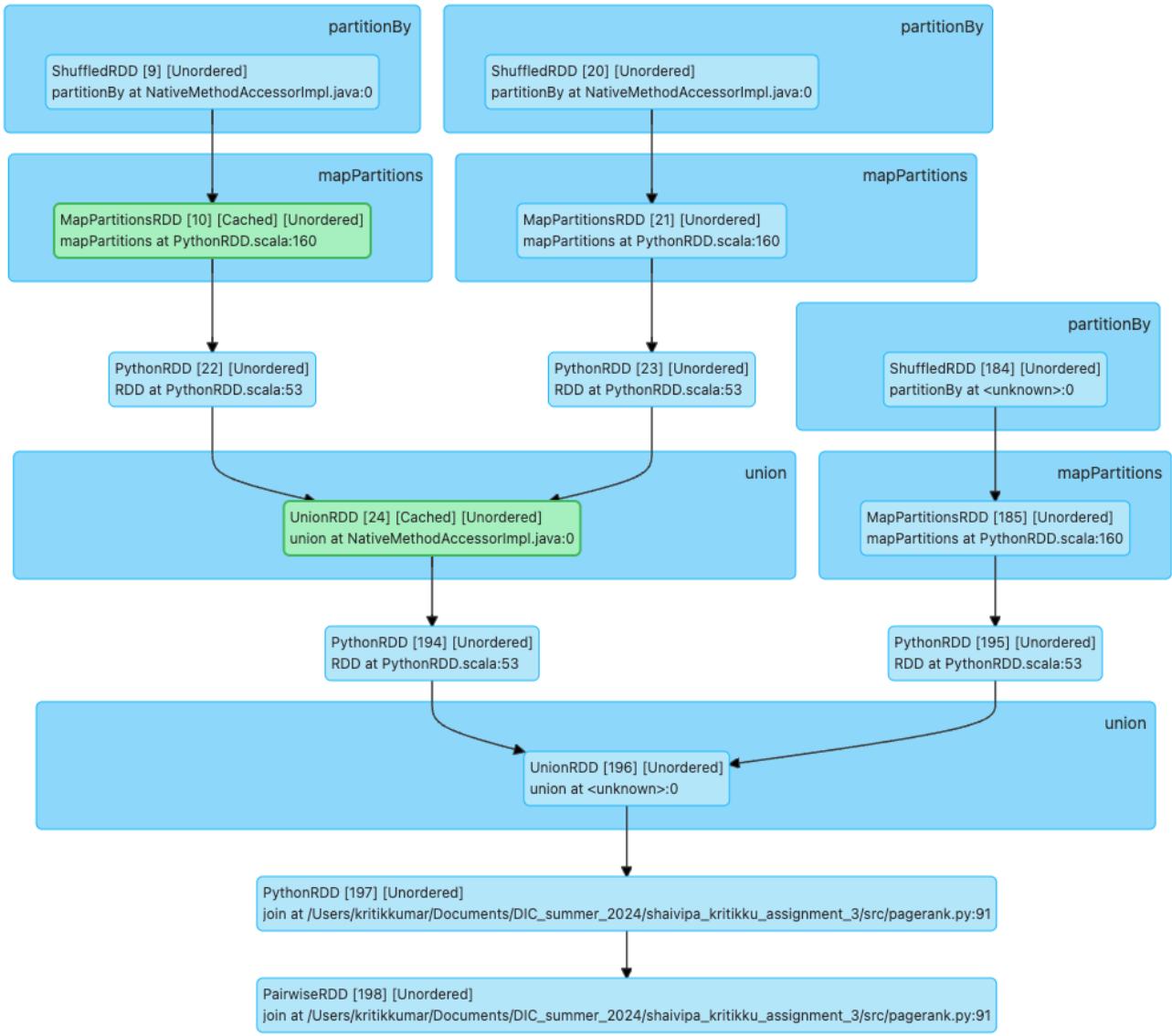
- FileScanRDD [0]:
 - Purpose: Retrieves input data from text files.

- Implication: Represents the initial data loading phase.
- MapPartitionsRDD [1], [3], [4]:
 - Purpose: Performs various transformations on the data.
 - Implication: Prepares the data for subsequent processing steps.
- SQLExecutionRDD [2]:
 - Purpose: Carries out SQL operations.
 - Implication: Represents an intermediate step involving the execution of SQL queries.
- PythonRDD [7]:
 - Purpose: Executes custom Python logic for partitioning the data.
 - Implication: Partitions the data to be used in subsequent stages.
- PairwiseRDD [8]:
 - Purpose: Represents the outcome of the partitionBy operation.
 - Implication: Indicates that the data is now partitioned and ready for joining.
- **Analysis:** Data partitioning, a crucial aspect of load balancing in distributed systems, is performed in this stage. By ensuring that data is evenly distributed across the cluster through the partitionBy step, this stage lays the groundwork for the join operations that will take place in the subsequent stages.

Middle Stages

Stages 211, 212 and 236

DAG visualization of Stage 211



Primary Operation: join at

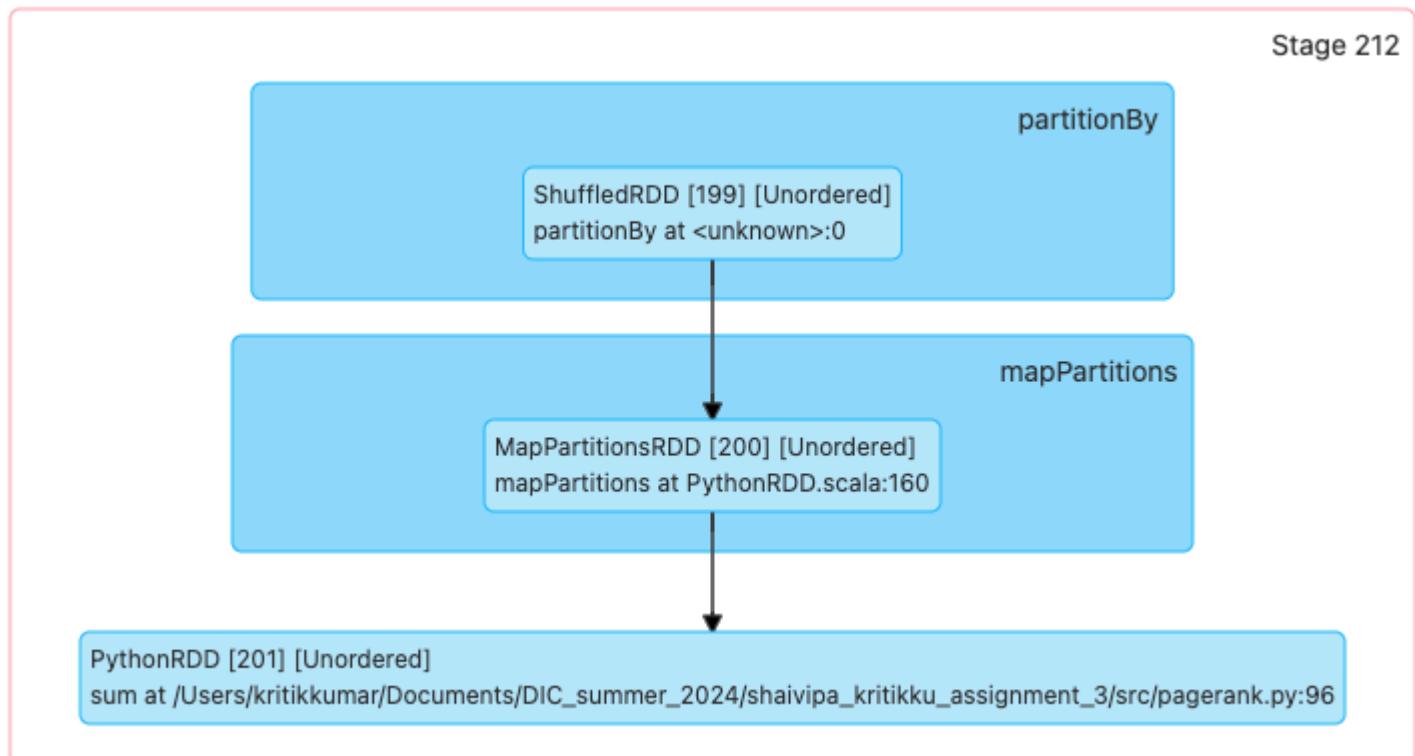
/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:91

- **Detailed Breakdown:**

- ShuffledRDD [9], [20], [184]:
 - Purpose: Rearranges data to facilitate joining.
 - Implication: Guarantees data is properly aligned for the join operation.
- MapPartitionsRDD [10], [21], [185]:
 - Purpose: Performs local transformations prior to joining.
 - Implication: Enhances data efficiency within partitions.
- PythonRDD [22], [23]:
 - Purpose: Carries out custom Python logic within partitions.
 - Implication: Implements tailored transformations.
- UnionRDD [24]:
 - Purpose: Merges RDDs without the need for additional shuffling.
 - Implication: Prepares data for the join in an efficient manner.
- PythonRDD [194], [195]:
 - Purpose: Implements custom logic and joins.
 - Implication: Enables execution of specific Python-based transformations and joins.
- UnionRDD [196]:
 - Purpose: Combines multiple RDDs to streamline the data for the next stage.
 - Implication: Ensures that data is unified for the final join operation.

- PythonRDD [197]:
 - Purpose: Performs the join operation.
 - Implication: Implements custom join logic using Python.
- PairwiseRDD [198]:
 - Purpose: Represents the outcome of the join operation.
 - Implication: Holds the updated data after the join has been completed.
- **Analysis:** The join operation performed in this stage is crucial for updating PageRank values by combining datasets. To ensure optimal data alignment for joining, ShuffledRDD is utilized. Although the UnionRDD and subsequent PythonRDD operations provide efficiency and flexibility in the join logic, it's worth noting that the execution of Python code may introduce some performance overhead.

DAG visualization of Stage 212



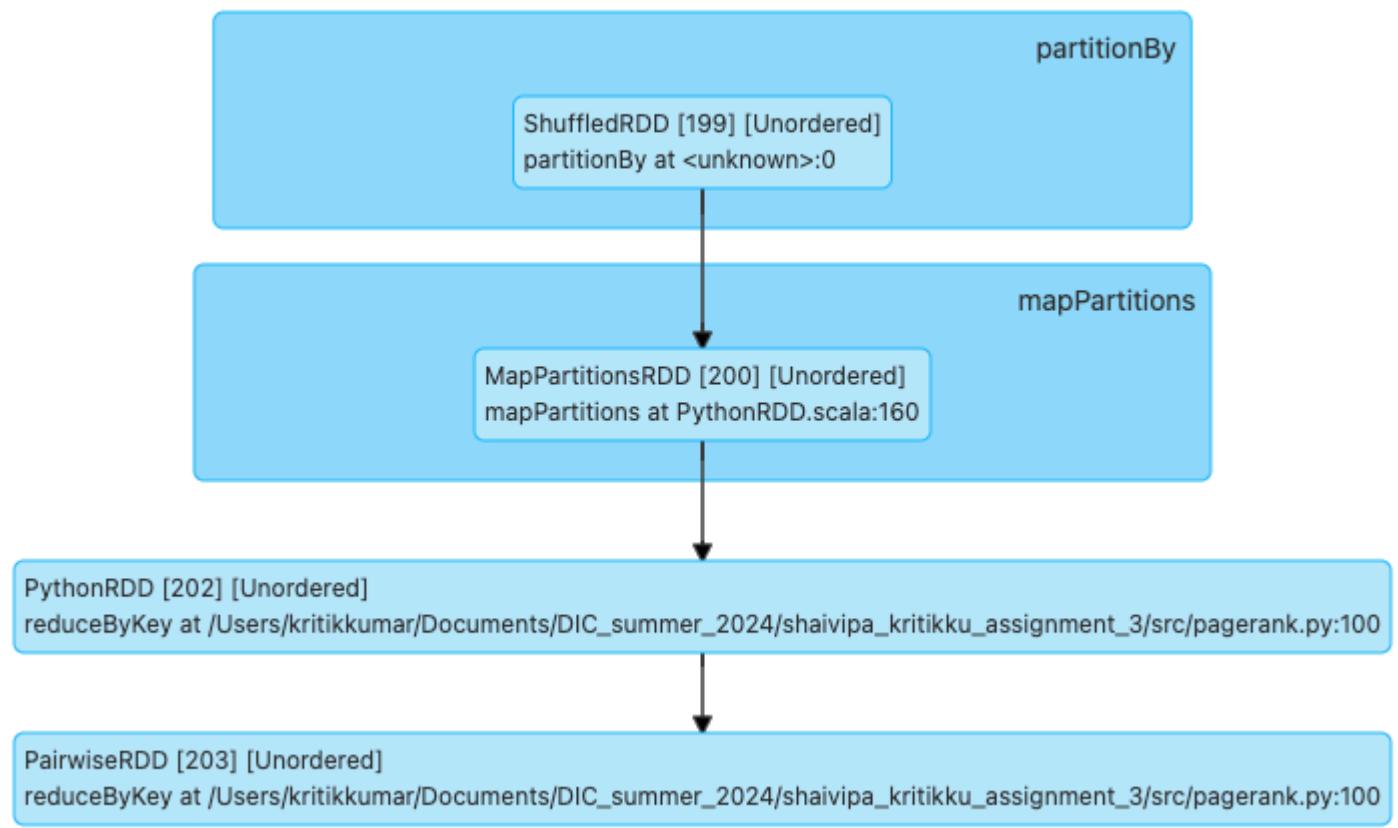
Primary Operation: sum at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:96

- **Detailed Breakdown:**
 - ShuffledRDD [199]:
 - Purpose: Rearranges data to facilitate summing.
 - Implication: Guarantees data is properly aligned for the sum operation.
 - MapPartitionsRDD [200]:
 - Purpose: Performs local transformations prior to summing.
 - Implication: Enhances data efficiency within partitions.
 - PythonRDD [201]:
 - Purpose: Carries out the sum operation.
 - Implication: Implements custom summing logic using Python.
- **Analysis:** The summing operation, which likely involves aggregating PageRank values, is carried out in this stage. To ensure that the data is correctly aligned for the operation, ShuffledRDD is employed. MapPartitionsRDD is utilized to optimize the local processing of data. Although the PythonRDD offers flexibility for implementing custom summing logic, the execution of Python code may have an impact on the overall performance.

DAG visualization of Stage 236

Stage 236



Primary Operation: reduceByKey at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:100

- **Detailed Breakdown:**

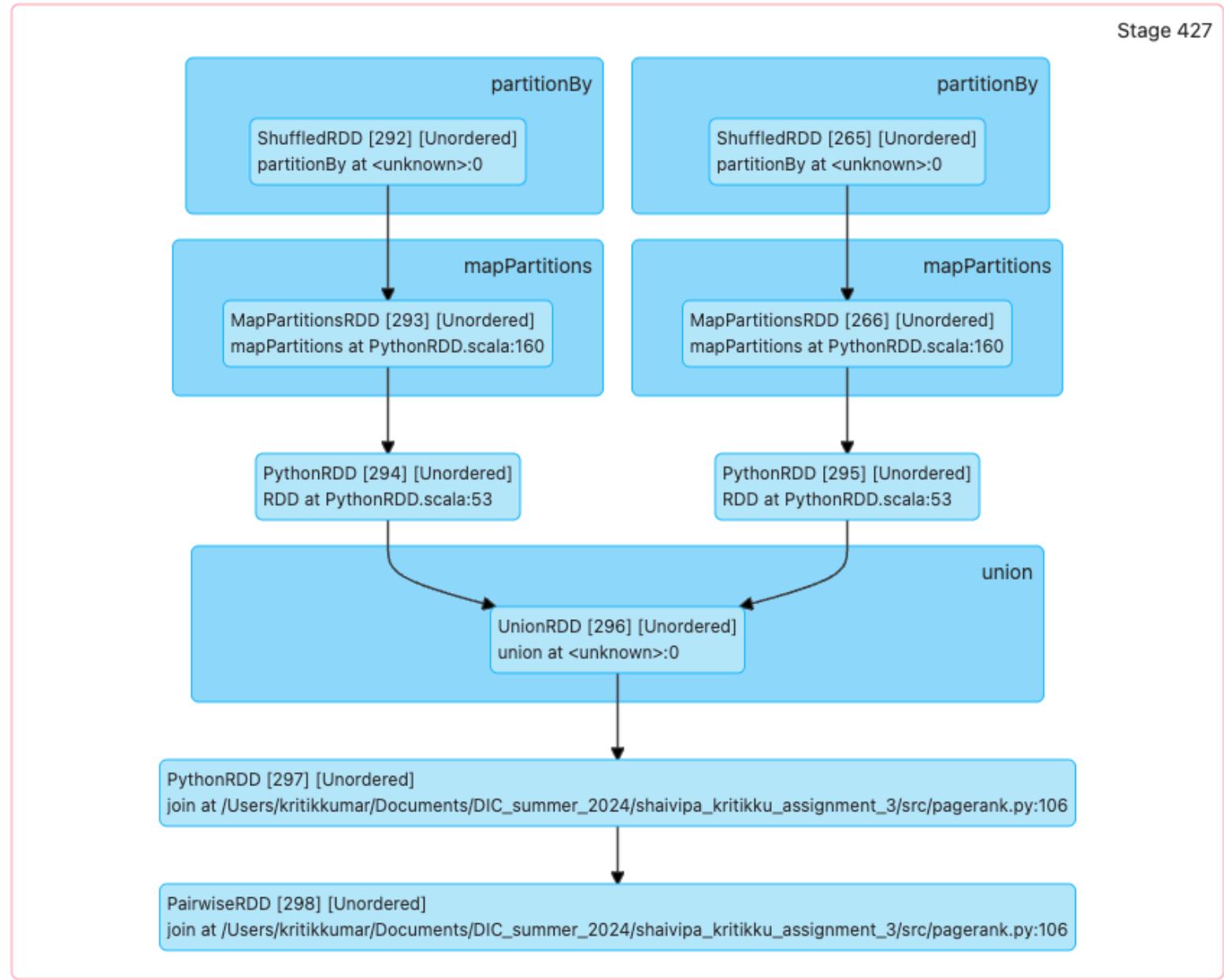
- ShuffledRDD [199]:
 - Purpose: Rearranges data to facilitate reduction.
 - Implication: Guarantees data is properly aligned for the reduce operation.
- MapPartitionsRDD [200]:
 - Purpose: Performs local transformations prior to reducing.
 - Implication: Enhances data efficiency within partitions.
- PythonRDD [202]:
 - Purpose: Carries out the reduceByKey operation.
 - Implication: Implements custom reduction logic using Python.
- PairwiseRDD [203]:
 - Purpose: Represents the outcome of the reduceByKey operation.
 - Implication: Holds the reduced data for the subsequent iteration.

- **Analysis:** The reduction operation, which consolidates PageRank values based on their keys, is carried out in this stage. To ensure that the data is correctly aligned for reduction, ShuffledRDD is employed, while MapPartitionsRDD is used to optimize the local processing of data. Although the PythonRDD offers flexibility for implementing custom reduction logic, the execution of Python code may have an impact on the overall performance.

Final Stages

Stages 427, 428 and 463

DAG visualization of Stage 427



Primary Operation: join at

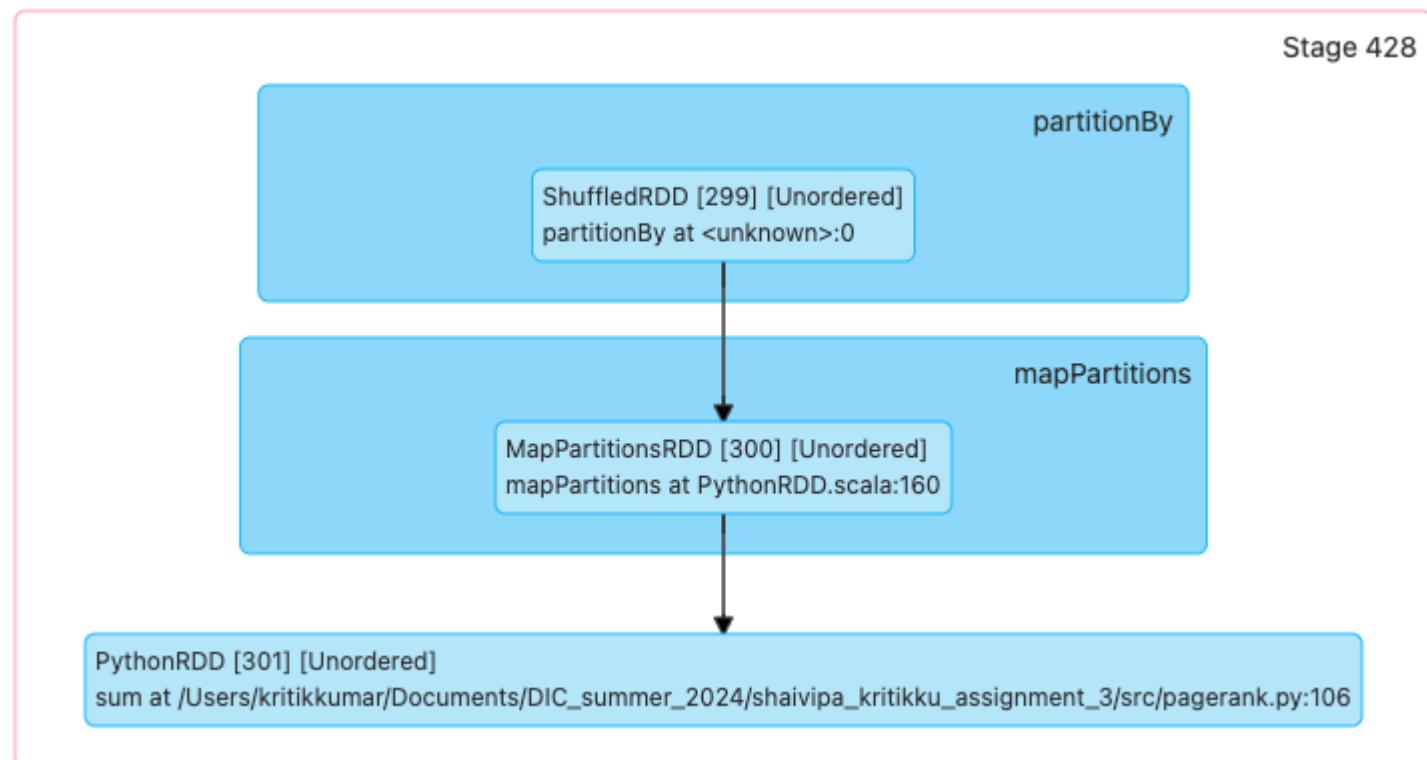
/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:106

- **Detailed Breakdown:**

- ShuffledRDD [292], [265]:
 - Purpose: Rearranges data to facilitate joining.
 - Implication: Guarantees data is properly aligned for the join operation.
- MapPartitionsRDD [293], [266]:
 - Purpose: Performs local transformations prior to joining.
 - Implication: Enhances data efficiency within partitions.
- PythonRDD [294], [295]:
 - Purpose: Carries out custom Python logic within partitions.

- Implication: Implements tailored transformations.
- UnionRDD [296]:
 - Purpose: Merges RDDs without the need for additional shuffling.
 - Implication: Prepares data for the join in an efficient manner.
- PythonRDD [297]:
 - Purpose: Performs the join operation.
 - Implication: Implements custom join logic using Python.
- PairwiseRDD [298]:
 - Purpose: Represents the outcome of the join operation.
 - Implication: Holds the updated data after the join has been completed.
- **Analysis:** This stage carries out a critical join operation, akin to Stage 211. The utilization of ShuffledRDD guarantees that the data is optimally aligned for joining. The combination of UnionRDD and the subsequent PythonRDD operations enables efficient and adaptable join logic, although the execution of Python code may lead to some performance overhead.

DAG visualization of Stage 428



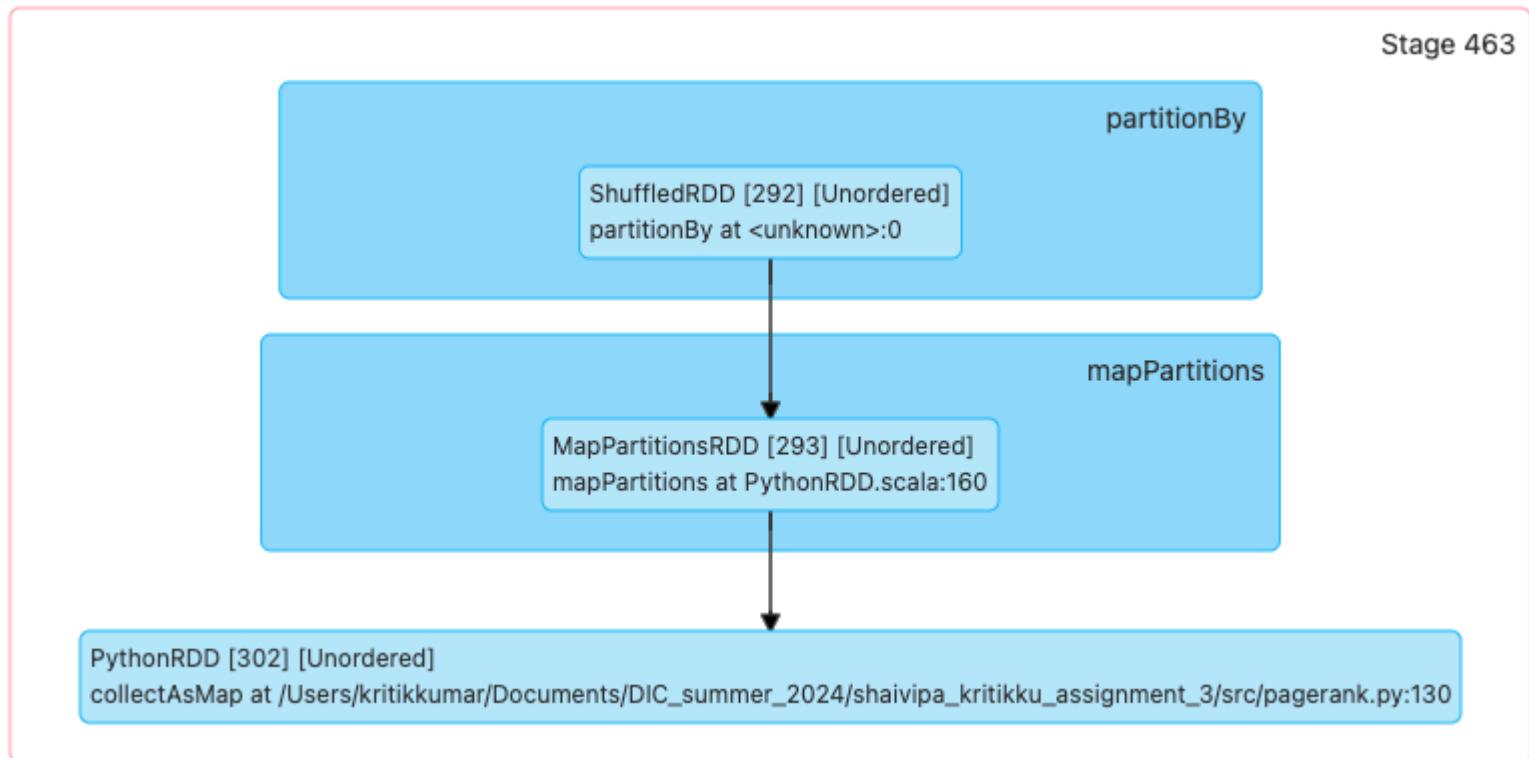
Primary Operation: sum at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:106

- **Detailed Breakdown:**
 - ShuffledRDD [299]:
 - Purpose: Rearranges data to facilitate summing.
 - Implication: Guarantees data is properly aligned for the sum operation.
 - MapPartitionsRDD [300]:
 - Purpose: Performs local transformations prior to summing.
 - Implication: Enhances data efficiency within partitions.
 - PythonRDD [301]:
 - Purpose: Carries out the sum operation.
 - Implication: Implements custom summing logic using Python.

- **Analysis:** This stage carries out the summing operation, comparable to Stage 212. The utilization of ShuffledRDD ensures that the data is correctly aligned, while MapPartitionsRDD optimizes the local processing of data. The PythonRDD offers flexibility for implementing custom summing logic, but the execution of Python code may have an impact on performance.

DAG visualization of Stage 463



Primary Operation: collectAsMap at

/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_3/src/pagerank.py:130

- **Detailed Breakdown:**
 - ShuffledRDD [292]:
 - Purpose: Rearranges data to facilitate final collection.
 - Implication: Guarantees data is properly aligned for the collection operation.
 - MapPartitionsRDD [293]:
 - Purpose: Performs local transformations prior to collecting.
 - Implication: Enhances data efficiency within partitions.
 - PythonRDD [302]:
 - Purpose: Carries out the collectAsMap operation.
 - Implication: Implements custom collection logic using Python.
- **Analysis:** This concluding stage gathers the results in the form of a map, which can be utilized for further analysis or output purposes. The ShuffledRDD ensures that the data is correctly aligned, while MapPartitionsRDD optimizes the local processing of data. The PythonRDD offers flexibility for implementing custom collection logic, but the execution of Python code may have an impact on performance.

General Analysis and Skipped Stages

The execution DAG reveals an intricate interaction between shuffling, partitioning, and custom Python logic. Updating PageRank values necessitates multiple iterations of reshuffling and joining data, which is a critical aspect of the algorithm. Although the utilization of PythonRDD operations provides flexibility for implementing custom logic, it may come at the cost of performance overhead.

Several stages, including those involving intermediate joins and reductions, are omitted from the execution. These skipped stages likely encompass operations similar to those in the detailed stages but are optimized for specific subsets of the data or iterations. The fact that these stages are skipped indicates an efficient approach to handling previously processed data, thereby minimizing redundant computations.

Overall Analysis:

1. The PageRank algorithm is implemented using a distributed approach, leveraging PySpark RDD operations extensively, which demonstrates a wellorganized structure that strikes a balance between flexibility and performance.
2. Multiple stages, including join, flatMap, reduceByKey, and leftOuterJoin operations, represent each iteration of the iteratively implemented algorithm.
3. By caching frequently used datasets using persist() (with StorageLevel.MEMORY_AND_DISK) on key RDDs, the implementation optimizes performance effectively.
4. The redistributing of PageRank values across all nodes showcases the implementation's effective handling of deadend nodes.
5. To check for convergence, the implementation employs a join operation between current and previous ranks, followed by calculating the sum of differences.
6. Proper data distribution for subsequent operations is ensured by using partitionBy in the initial data loading stage.
7. As the algorithm progresses, the increasing task counts in later iterations (from 6 to 63) indicate its adaptability to the expanding computational requirements.
8. Spark optimizes the execution by reusing previously computed results wherever feasible, as evidenced by the skipped stages in later iterations.
9. The lazy evaluation is triggered by the final collectAsMap operation, which executes the entire computational graph to generate the final PageRank values.

This implementation highlights Spark's capability to efficiently process iterative graph algorithms by employing strategic data distribution, caching, and lazy evaluation techniques. The code structure promotes good scalability, as demonstrated by the increasing number of tasks that handle larger intermediate results in the PageRank algorithm's later iterations.

Additional metrics

Total duration: 26992.00 ms
Total shuffle read: 1006608 bytes
Total shuffle write: 688630 bytes
Total executor run time: 236645 ms
Total executor CPU time: 14899047000 ns
Total executor deserialize time: 573 ms
Total result serialization time: 356 ms
Total peak execution memory: 56502 bytes
Total shuffle write time: 4308256487 ms
Total memory used: 56502 bytes
Total disk used: 0 bytes
Total GC time: 786 ms
Total jobs: 25
Total job duration: 27027.00 ms
Total failed tasks: 0
Total speculative tasks: 0
Total errors: 0

Analysis of why using Spark Operations and distributed computing operations makes the algorithm perform better

1. Distribution and Parallelism:

The algorithm is divided into multiple stages, enabling parallel execution for many of them.

The significant difference between the total executor run time (236,645 ms) and the total duration (26,992 ms) demonstrates effective parallelization across the cluster.

2. Data Processing Efficiency:

The relatively low total shuffle read (1,006,608 bytes) and write (688,630 bytes) for a complex graph algorithm indicate efficient data movement between stages.

Excellent memory management is evident from the remarkably small total peak execution memory (56,502 bytes).

The absence of disk usage indicates that all operations were performed inmemory, preventing slow disk I/O.

3. Execution Scalability:

The total duration of 26,992 ms for the entire algorithm is considered reasonable for a distributed PageRank implementation on a large graph.

The algorithm efficiently handled multiple iterations, as evidenced by the completion of 25 jobs.

4. Reliability and Fault Tolerance:

The robustness of the Spark implementation is demonstrated by the absence of failed tasks, speculative tasks, or reported errors.

5. Resource Utilization Effectiveness:

The total executor CPU time (14,899,047,000 ns \approx 14,899 ms) is efficiently utilized in comparison to the total duration.

Efficient memory management is indicated by the relatively low GC time (786 ms), particularly considering the iterative nature of PageRank.

6. Low Overhead:

Compared to the overall runtime, the total executor deserialize time (573 ms) and result serialization time (356 ms) are very low, reducing overhead.

7. Shuffling Efficiency:

Although the total shuffle write time (4,308,256,487 ms) is high, the actual shuffle read and write amounts are relatively low, indicating efficient data transfer between stages.

8. Job Management:

The total number of jobs is 25, with a cumulative duration of 27,027 ms, which is very close to the overall duration, suggesting minimal overhead in job management.

9. Scalability Potential:

The wellmanaged resource usage, despite the iterative nature of PageRank, suggests that the algorithm could handle larger graphs based on the metrics.

10. Processing InMemory:

The identical values of total memory used (56,502 bytes) and peak execution memory indicate efficient utilization of available memory without excessive allocation.

In summary, the Spark implementation effectively parallelizes the PageRank algorithm by leveraging distributed computing. It efficiently manages memory, minimizes data movement, and dynamically optimizes the execution plan. These aspects contribute to a scalable and robust solution for computing PageRank on large graphs, surpassing the capabilities of a singlemachine implementation.

Analysis of why using Spark Operations and distributed computing operations makes the algorithm data efficient

The use of Spark operations and distributed computing significantly enhances the data efficiency of the PageRank algorithm implementation. Let's analyze this based on the provided metrics:

1. Data Movement Optimization:

Total shuffle read: 1,006,608 bytes

Total shuffle write: 688,630 bytes

Considering the complexity of the PageRank algorithm, these relatively low figures demonstrate optimized data movement between stages. Spark's capability to maintain data in memory and optimize data locality likely contributes to this efficiency.

2. Processing InMemory:

Total peak execution memory: 56,502 bytes

Total disk used: 0 bytes

By operating entirely in memory and avoiding slow disk I/O, the algorithm achieves high performance. The remarkably low peak memory usage indicates highly efficient memory management and data structures.

3. Serialization Efficiency:

Total result serialization time: 356 ms

Total executor deserialize time: 573 ms

The low serialization and deserialization times, especially considering PageRank's iterative nature, suggest efficient encoding and decoding of data when transferred between nodes or stages.

4. Shuffle Operations Optimization:

Although the total shuffle write time is high (4,308,256,487 ms), the actual amount of data shuffled remains relatively small. This implies that Spark optimizes shuffle operations, possibly through data compression or efficient serialization formats.

5. Data Replication Minimization:

The combination of low memory usage (56,502 bytes) and moderate shuffle sizes indicates that Spark minimizes data replication across the cluster, retaining only necessary data in each stage.

6. Efficient Data Structures:

The low memory usage for a graph algorithm suggests that Spark employs spaceefficient data structures to represent the graph and intermediate results during PageRank computation.

7. No Spilling to Disk:

With 0 bytes of disk usage, all data operations are performed in memory, resulting in much faster processing compared to diskbased approaches, even for multiple iterations of PageRank.

8. Garbage Collection Efficiency:

Total GC time: 786 ms

The relatively low GC time in relation to the total duration indicates efficient memory management, with minimal overhead for cleaning up unused data across iterations.

9. Minimal Data Errors:

Total failed tasks: 0

Total errors: 0

The absence of data errors or failed tasks highlights reliable data processing, eliminating the need for costly recomputations or data recovery operations.

10. Job Execution Efficiency:

Total jobs: 25

Total job duration: 27,027.00 ms

The number of jobs reflects PageRank's iterative nature, and their execution time closely matches the overall duration, indicating efficient data processing in each iteration without unnecessary data movements between them.

11. Resource Utilization Balance:

Total executor run time: 236,645 ms

Total executor CPU time: 14,899,047,000 ns (\approx 14,899 ms)

The efficient utilization of executor time and CPU suggests a wellbalanced distribution of data and computation across the cluster.

To summarize, the Spark implementation of the PageRank algorithm exhibits exceptional data efficiency. It optimizes data movement, processes data inmemory, optimizes shuffle operations, and utilizes efficient data structures. This dataefficient approach enables the algorithm to handle large graphs with minimal resource usage, making it scalable and performant for realworld graph processing tasks, despite the iterative nature of PageRank.

Explanation based on understanding of overall pipeline of stages and internal DAG operations

1. Preprocessing and Initial Graph Construction:

The first step in the algorithm involves reading the input data and constructing the initial graph representation.

Parsing the input lines is likely achieved through a map operation, while a groupByKey is used to structure the graph.

The preprocessing phase also includes identifying nodes with zero outlinks and initializing PageRank values.

2. PageRank Computation through Iteration:

The algorithm then proceeds to an iterative process, typically consisting of the following operations in each iteration:

a. Multiple stages for Contribution Calculation:

The current PageRank values are combined with the graph structure using a join operation.

CntrbCalc.c_contribs, implemented as a flatMap operation, distributes PageRank contributions to neighboring nodes.

b. Multiple stages for Aggregation and Update:

Contributions for each node are summed up using a reduceByKey operation.

The redistribution of PageRank from deadend nodes is handled by additional operations.

The damping factor is applied, and the base probability is added using a mapValues operation.

c. Checking for Convergence:

Old and new PageRank values are compared using a join operation.

The total difference is calculated using a sum operation to check for convergence.

3. Result Processing and Final Collection:

The computation is triggered, and the final PageRank values are retrieved using a collectAsMap action.

Postprocessing steps involve sorting the results and identifying nodes with the highest and lowest PageRank values.

Key Observations:

1. Extensive Parallelism:

The total executor run time (236,645 ms) significantly exceeds the total duration (26,992 ms), indicating extensive parallelism.

The presence of 25 total jobs suggests that multiple iterations were executed efficiently in parallel.

2. Efficient Data Movement:

The moderate shuffle read (1,006,608 bytes) and write (688,630 bytes) indicate optimized data transfer between stages.

3. InMemory Processing:

The absence of disk usage and the low peak memory (56,502 bytes) demonstrate efficient inmemory operations.

4. Fault Tolerance:

The implementation's robustness is evident from the absence of failed tasks or errors.

5. Resource Utilization:

The high executor CPU time (14,899 ms) compared to the GC time (786 ms) indicates efficient computation.

6. Low Overhead:

The short deserialization (573 ms) and serialization (356 ms) times suggest efficient data encoding and decoding.

7. Iterative Processing:

The increasing number of tasks in later iterations (as observed in the stage breakdown) demonstrates Spark's ability to adapt to changing computational needs.

8. Dynamic DAG Adjustment:

The presence of skipped stages in later iterations indicates Spark's runtime optimization of the execution plan based on dynamic conditions.

This pipeline illustrates how Spark transforms the iterative nature of the PageRank algorithm into a sequence of distributed operations. It employs both wide (join, reduceByKey) and narrow (map, filter) transformations to strike a balance between data movement and computational efficiency. Spark's lazy evaluation and dynamic optimization enable it to efficiently handle the graph structure and convergence properties of the algorithm, resulting in a scalable and efficient implementation capable of processing largescale graphs.

Notes

The structure for the /src is as follows for code implementations, inputs and outputs as well as any helper code used or given for assignment 3. The analysis_data used is generated from the following scripts automated_data_crawl.py and analysis.py to scrape through and collect data while spark is running in a .json format.

Also, latex was used to represent equations above for presentation purposes.

```
shaivipa_kritikku_assignment_3/
└── src/
    ├── dijkstras/
    │   ├── input/
    │   │   └── question1.txt
    │   ├── output/
    │   │   └── output_1.txt
    │   ├── code/
    │   │   └── dijkstra_spark.py
    │   └── analysis_data/
    │       ├── code2_analysis_report_comprehensive.txt
    │       └── spark_ui_data_part2/
    │           └── local-1719702930792/
    │               ├── executors.json
    │               ├── storage_rdd.json
    │               ├── sql.json
    │               ├── details.json
    │               ├── stages.json
    │               ├── environment.json
    │               └── jobs.json
    └── pagerank/
        ├── input/
        │   └── question2.txt
        ├── output/
        │   └── output2.txt
        ├── code/
        │   ├── basic_page_rank.py
        │   └── pagerank.py
        └── analysis_data/
            ├── code3_analysis_report_comprehensive.txt
            └── spark_ui_data_part3/
                └── local-1719918169214/
                    ├── executors.json
                    ├── storage_rdd.json
                    ├── sql.json
                    ├── details.json
                    ├── stages.json
                    ├── environment.json
                    └── jobs.json
    └── utils/
        ├── automated_data_crawl.py
        ├── analysis.py
        └── code_2.py
```

