ASSIGNMENT-3 REPORT

Aditya Srivatsav Lolla(alolla |50559685)
Sai Krishna Goud Valdas(svaldas | 50560726)

PART -1: Review paper on Hadoop/HDFS .

Review of "Hadoop Distributed File System"

An extensive overview of the Hadoop Distributed File System (HDFS) is provided in the publication "Hadoop Distributed File System" by Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The architecture, design tenets, and practical applications of HDFS as they relate to Yahoo!'s massive data processing are highlighted in this paper.

Introduction and Related Work:
Setting HDFS in the larger Hadoop ecosystem—which consists of elements like MapReduce, HBase, Pig, Hive, ZooKeeper, Chukwa, and Avro—is the first thing the writers accomplish. They highlight how Hadoop's ability to divide data and computation over numerous hosts makes it easier to execute parallel processing in close proximity to the data. Scalability is essential for effectively and financially managing enormous volumes of data. The scalability and robustness of Hadoop is demonstrated in the introduction by mentioning that Yahoo!'s Hadoop clusters span over 25,000 computers and manage 25 petabytes of application data.

Architecture:
The NameNode and DataNodes, the two fundamental HDFS components, are explored in detail in the architecture section. The NameNode controls client access to files and maintains the filesystem namespace. It keeps track of metadata and how file blocks are mapped to DataNodes, which hold the actual data. To guarantee dependability and data availability, each file is split up into blocks, each of which is normally 128 MB in size. These blocks are then replicated over several DataNodes. This replication approach allows data access from numerous nodes at the same time, which improves fault tolerance and boosts data availability.

Performance and Benchmarks:
HDFS's performance is crucial, and this article describes the benchmarks that are used to assess its I/O bandwidth. The DFSIO benchmark, which gauges throughput for read, write, and append operations devoid of external application interference, is described by the authors. These benchmarks' results show that HDFS provides notable I/O performance, especially in big clusters. The DFSIO benchmark, for instance, demonstrated an average read throughput of 66 MB/s and write throughput of 40 MB/s per node. In a production setting, throughput is inherently reduced but nevertheless important because of competition from other applications.

Reliability and Fault Tolerance:
The design of HDFS places a strong emphasis on fault tolerance and dependability. Several strategies that guarantee data availability and integrity are described in the paper. One main method to deal with node failures is to replicate data blocks across several DataNodes. To find and fix irregularities early on, the system also uses routine block scanning and reporting. When faults occur, HDFS's ability to efficiently recover and re-replicate data blocks adds to its already impressive robustness.

Conclusion:
In closing, the paper offers a reflection on Yahoo!'s real-world experience with HDFS. Its scalability, dependability, and capacity to manage complex data processing jobs are highlighted by the authors. Additionally, they discuss the current and next developments for HDFS, including ways to increase system efficiency and the NameNode's capacity to manage a bigger namespace.

In conclusion, the paper "Hadoop Distributed File System" offers a thorough examination of HDFS, highlighting its reliability characteristics, performance capabilities, and architectural design. It is an invaluable tool for learning how to design and manage large-scale data processing systems efficiently.

PART-2:
IMPLEMENTATION :
→ The provided code reads a graph from a file and implements Dijkstra's algorithm to find the shortest path and its length between a source and a target node.

```python
import heapq


def read_graph_from_file(filename):
    graph = {}
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split(',')
            source = int(parts[0].strip())
            target = int(parts[1].strip())
            weight = int(parts[2].strip())

    return graph

def dijkstra_length_and_path(graph, source, target):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]

    paths = {source: []}

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_node == target:
            break

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
```

```
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
                paths[neighbor] = paths[current_node] + [current_node]

    if distances[target] == float('inf'):
        return [], float('inf')

    path = paths[target] + [target]

    return path, distances[target]
```

→ Reads a graph from the specified file and prints the graph dictionary and writes the shortest paths and distances from node 0 to all other nodes in the graph to an output file using Dijkstra's algorithm.

```
[77] graph = read_graph_from_file("/content/drive/MyDrive/Colab Notebooks/DIC Assignment-3/question1.txt")
     print(graph)
```

```
[74]
     f = open("output_1.txt", "w")
     for i in range(1,len(graph)):
       path, distance = dijkstra_length_and_path(graph, 0, i)
       f.write(f"Shortest path from node 0 to node {i}: {path}\n")
       f.write(f"Total distance: {distance}\n")
     f.close()
```

OUTPUT : These results show the shortest paths and corresponding total distances from node 0 to nodes 1, 2, 3, and 4, calculated using Dijkstra's algorithm.

output_1.txt

```
1 Shortest path from node 0 to node 1: [0, 1]
2 Total distance: 2
3 Shortest path from node 0 to node 2: [0, 2]
4 Total distance: 3
5 Shortest path from node 0 to node 3: [0, 1, 4, 3]
6 Total distance: 4
7 Shortest path from node 0 to node 4: [0, 1, 4]
8 Total distance: 3
9
```

Part-3 Implement and analyze Page-Rank algorithm.

→ Creates a graph, computes and prints the PageRank of each node, visualizes the graph, and identifies nodes with the highest and lowest PageRank.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode

def main(filename):
    spark = SparkSession.builder \
        .appName("PageRank Example") \
        .getOrCreate()


    edges = spark.read.text(filename) \
        .rdd.map(lambda row: row.value.split(': ')) \
        .map(lambda x: (int(x[0]), list(map(int, x[1].strip('[]').split(', '))))) \
        .flatMapValues(lambda x: x) \
        .map(lambda x: (x[0], x[1]))


    edges_df = edges.toDF(["src", "dst"])


    from graphframes import GraphFrame
    vertices = edges_df.selectExpr("src as id").distinct()
    g = GraphFrame(vertices, edges_df)


    results = g.pageRank(resetProbability=0.15, maxIter=10)


    max_pagerank = results.vertices.orderBy('pagerank', ascending=False).first()
    min_pagerank = results.vertices.orderBy('pagerank', ascending=True).first()

    print(f"Highest PageRank: Node {max_pagerank['id']} with rank {max_pagerank['pagerank']:.4f}")
    print(f"Lowest PageRank: Node {min_pagerank['id']} with rank {min_pagerank['pagerank']:.4f}")

    spark.stop()

if __name__ == "__main__":
    main('/content/drive/MyDrive/Colab Notebooks/DIC Assignment-3/question2.txt')
```

OUTPUT :

```
(anaconda3) (base) adityasrivatsav@Adityas-A
Highest PageRank: Node 79 with rank 0.0278
Lowest PageRank: Node 20 with rank 0.0026
```

Dijkstra’s Algorithm DAG Visulaization:

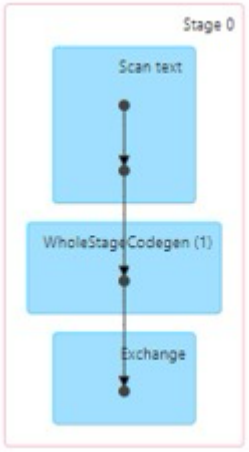
Description of Dijkstra's Algorithm:

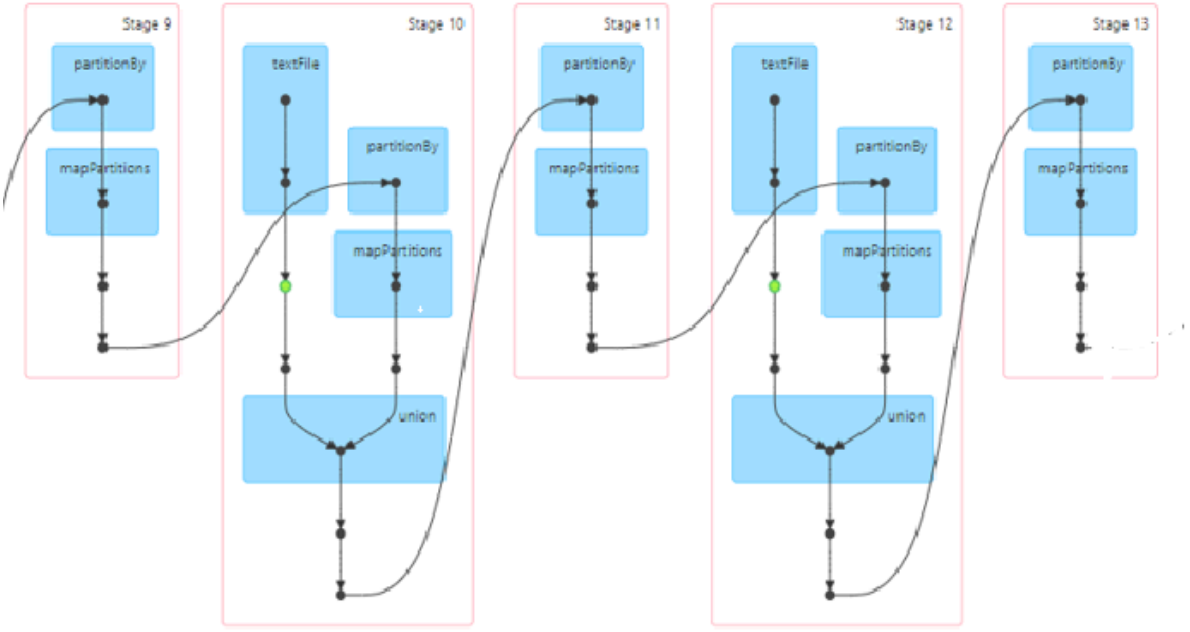
Dijkstra's Algorithm is a famous algorithm used for finding the shortest paths between nodes in a graph. It is widely used in network routing protocols and geographical mapping.

- The image shows stages from 9 to 13.
- Each stage contains tasks like partitionBy, mapPartitions, union, and textFile.
- The data flow and transformations are visualized with arrows connecting different operations across stages.
- The image shows Stage 0.
- It includes operations like Scan text, WholeStageCodegen, and Exchange.
- The tasks are executed sequentially, showing the initial processing steps.

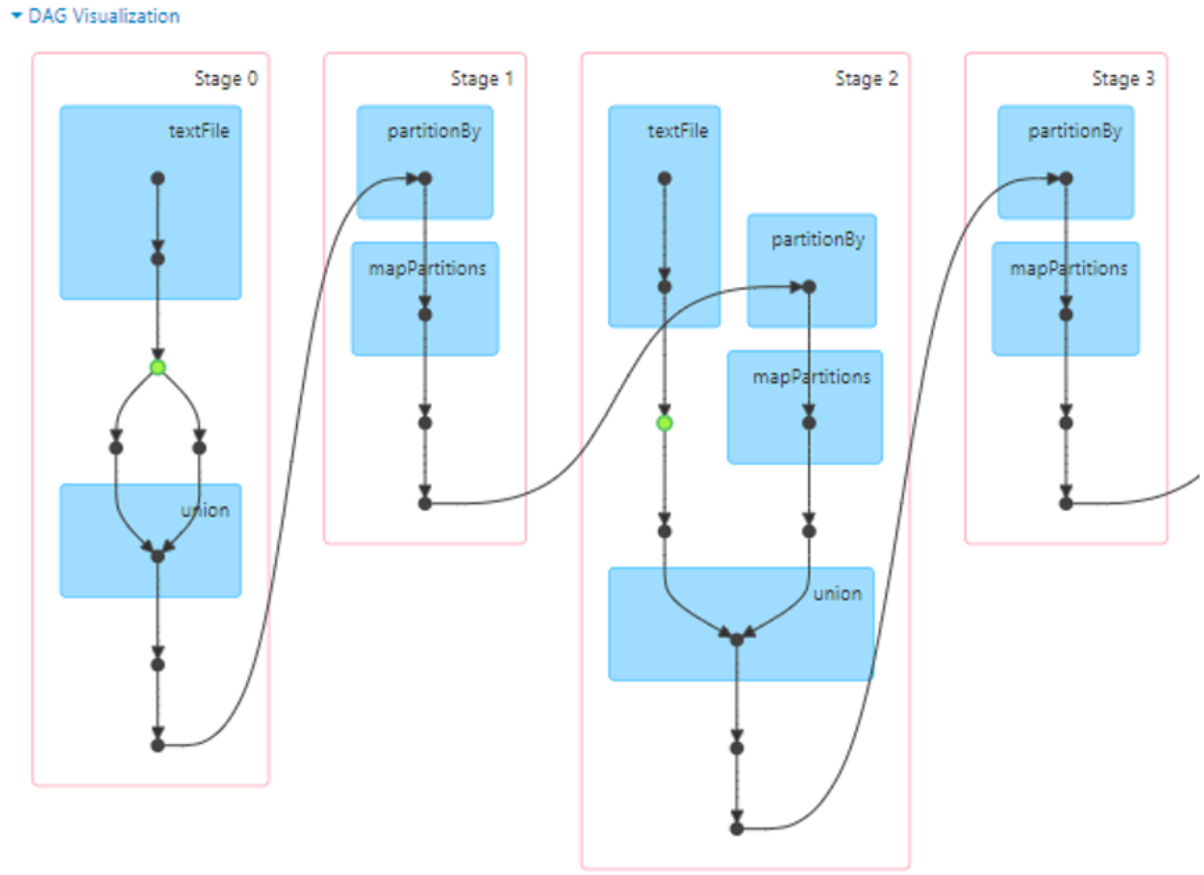
Steps of Dijkstra's Algorithm:

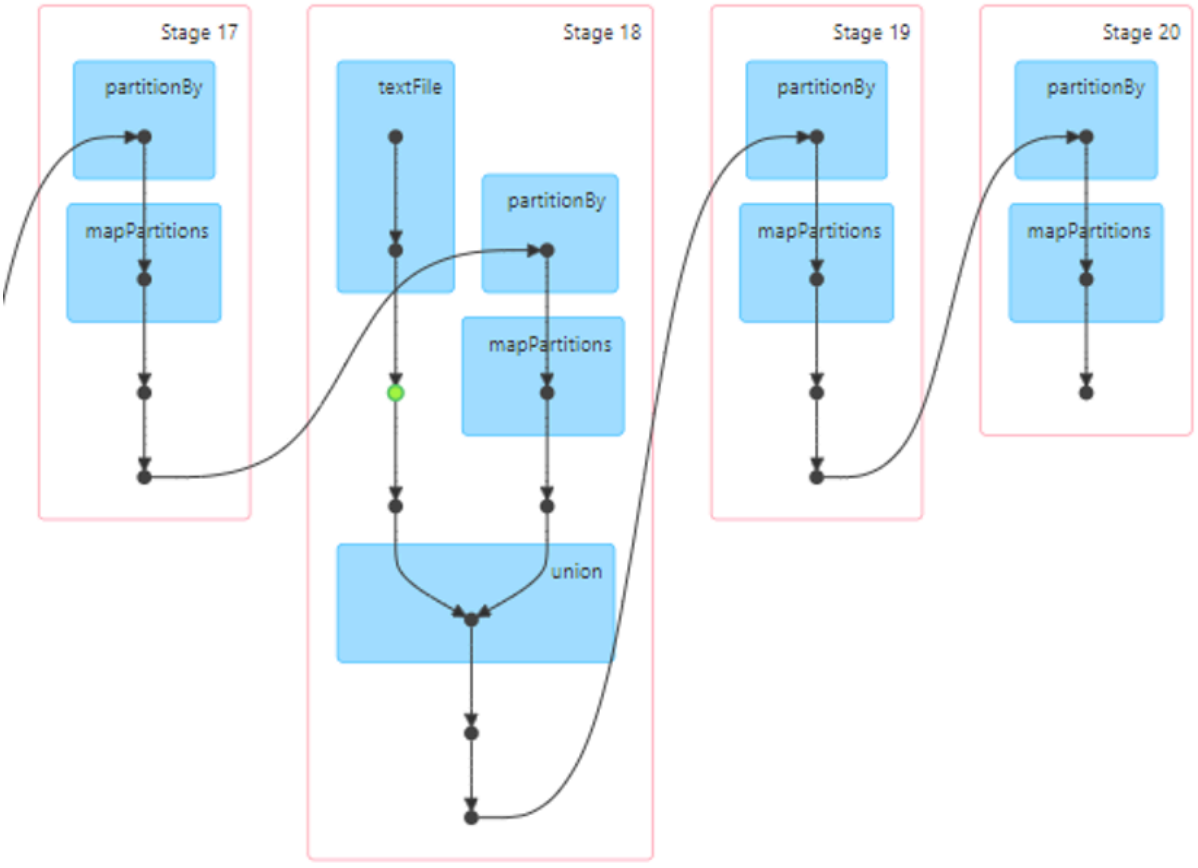
1. **Initialization:**
    - Start with the initial node (source) and set its distance to 0. Set the distance to all other nodes to infinity.
    - Create a priority queue to hold all nodes in the graph, prioritized by their current shortest distance.
2. **Processing Nodes:**
    - Extract the node with the smallest distance from the priority queue (initially the source node).
    - For the current node, examine all its neighboring nodes.
    - Calculate the tentative distance of each neighboring node by summing the current node's distance and the edge weight to the neighbor.
    - If the calculated distance is less than the known distance, update the shortest distance and update the priority queue.
3. **Update Paths:**
    - Keep track of the path to each node for reconstruction of the shortest path.
4. **Completion:**
    - Repeat the process until the priority queue is empty or until the shortest path to the destination node is found.
    - The algorithm guarantees that the shortest path is found for graphs with non-negative edge weights.


Stage 0
Scan text
WholeStageCodegen (1)
Exchange


Stage 9
partitionBy
mapPartitions
Stage 10
textFile
partitionBy
mapPartitions
union
Stage 11
partitionBy
mapPartitions
Stage 12
textFile
partitionBy
mapPartitions
union
Stage 13
partitionBy
mapPartitions

Page Rank DAG Visualization:





Stage 0: Initial Data Union

Description: This stage involves reading and combining data from text files, serving as the initial preparation step where input data is loaded into the Spark environment.

Operations: The main operation here is the union of text files, which consolidates data into a single RDD (Resilient Distributed Dataset) or DataFrame. This sets the foundation for subsequent transformations.

Stages 1, 3, 5, 7, ..., 19: Partitioning and Mapping

Description: These stages feature repetitive transformations focused on partitioning data and applying map functions to each partition. These stages are essential in the PageRank algorithm, where rank values are iteratively updated.

Operations: The key operations include:

- Repartitioning data to evenly distribute the workload across nodes.
- Applying map functions to each partition to update PageRank scores based on links and previous scores.

Stages 2, 4, 6, 8, ..., 18: Intermediate Processing

Description: These stages are similar to Stage 0 but occur repeatedly throughout the process. They involve reading data, potentially to refresh or update the dataset based on new PageRank computations.

Operations: Important operations during these stages include:

- Reading intermediate data or recalculating certain data points.
- Repartitioning and applying map functions as in other transformation stages.
- Performing union operations to consolidate results, preparing for the next iteration.

Stage 20: Final Preparation

Description: This final stage focuses on preparing the ultimate output of the PageRank algorithm after all iterations are complete. It ensures that the final PageRank values are correctly partitioned and ready for output or further processing.

Operations: The primary tasks in this stage involve:

- A final partitioning to optimize data layout for output.
- Application of map partitions to finalize the computations of PageRank values.