CS744: BIG DATA SYSTEMS

ASSIGNMENT 1

Zhiwei Song, Shahab Ahmad Khan, Varshita Rodda, Yash Chaudhari

Group 13

Overview

This report details the completion of Assignment 1, focusing on deploying Apache Hadoop as the underlying file system and Apache Spark as the execution engine. The assignment required the development of various Spark applications and a comprehensive understanding of their interactions with HDFS. Our group completed the tasks outlined in the assignment guidelines, including software deployment, a simple Spark application, and implementation of the PageRank algorithm. Additionally, we investigated custom DataFrame/RDD partitioning, in-memory persistence, and simulated worker failure scenarios.

Environment Setup

We completed the assignment in CloudLab, utilizing the provided profile "cs744-sp24-assignment1" under the "UWMadison744-S24" project. The profile consisted of a 3-node cluster with Ubuntu installed on each machine. We followed the provided instructions to set up Hadoop and Spark on the cluster, including mounting disks, deploying Hadoop, and configuring Spark.

Introduction

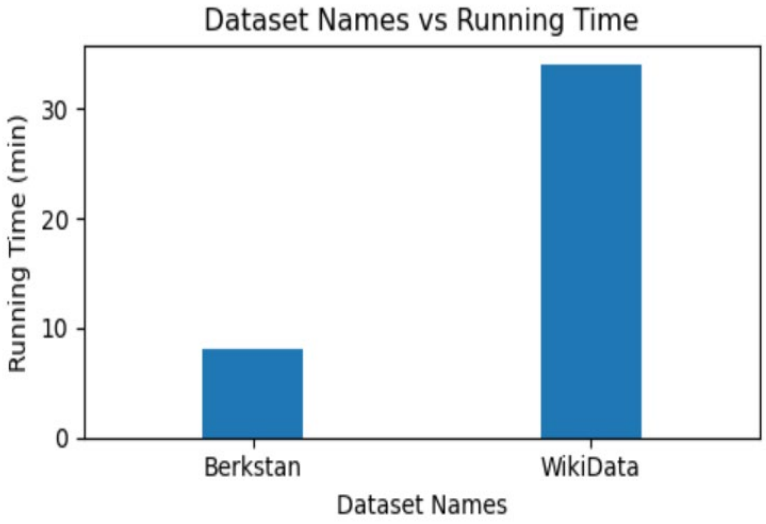
We successfully deployed Apache Hadoop and Apache Spark on the CloudLab cluster following the provided instructions. This involved downloading, configuring, and starting HDFS and Spark services on each node in the cluster. Utilizing PySpark, we implemented a Spark application to sort data based on country code and timestamp. The application loaded data into HDFS, processed it using Spark DataFrame operations, and stored the sorted output back into HDFS in csv format.

For Part 3, We implemented the PageRank algorithm using PySpark, executing it on two datasets: the Berkeley-Stanford web graph and the enwiki-20180601-pages-articles dataset. We iterated the algorithm for 10 iterations and analyzed its performance on the provided datasets.

Analysis and observations:

i. **Task 1: PageRank Algorithm Implementation**

Our group developed a PySpark application to implement the PageRank algorithm. We efficiently computed PageRank scores for the given datasets and observed their convergence over multiple iterations.



ii Task 2: Custom DataFrame/RDD Partitioning

We explored custom partitioning techniques for DataFrame and RDD in PySpark. By adjusting partitioning configurations, we analyzed changes in data distribution and parallelism, optimizing job performance where possible.

As per figure 1, We observed that the execution time remained consistent for Web-Berkstan dataset (smaller dataset ~128mb) despite increasing the number of RDD partitions. We attribute this consistency to the relatively small size of the input data.

For wiki-data, figure 1(b) illustrates that initially, as the number of RDD partitions increases, the runtime decreases. This trend persists until the number of RDD partitions reaches 90. Beyond this point, we observe a plateau where the runtime remains relatively constant despite further increases in the number of RDD partitions. Further experimentation with increasing partitions up to 1000 may provide additional insights into the runtime behavior.

iii Task 3: In-Memory Persistence

We persisted appropriate DataFrame/RDD objects in memory using PySpark's caching mechanism.

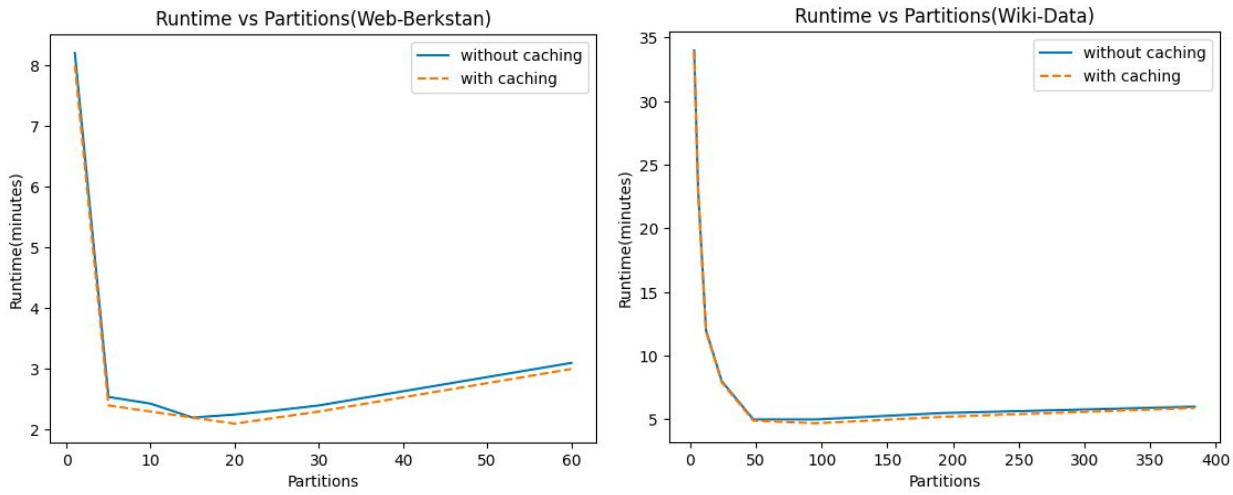
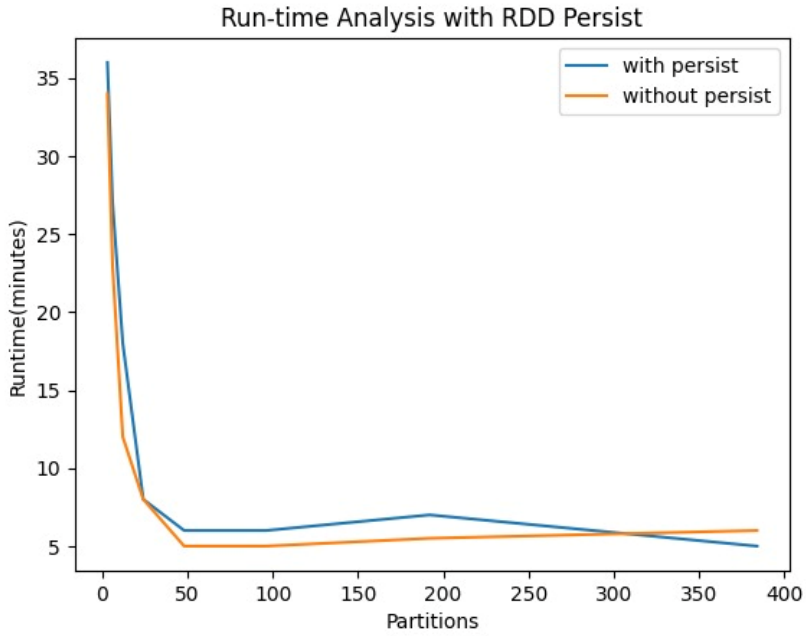


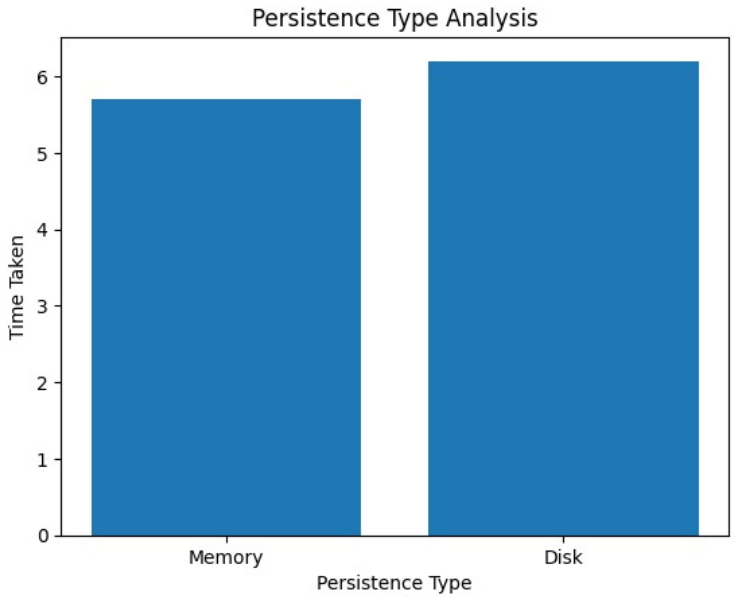
Figure 1: Analysis of runtime with varying number of partitions for the two datasets

The graph(figure 1) compares the execution time under two conditions: one with cache enabled and the other disabled. Initially, increasing the number of partitions results in a decrease in total execution time due to the advantages of higher concurrency. However, as shown in Figure, the execution time exhibits an unexpected increase with a larger number of partitions. We attribute this phenomenon to the amplified overhead in task scheduling and network transmission.



Initially, we anticipated experiencing performance improvements by persisting the input Resilient Distributed Dataset (RDD). However, upon closer observation, we encountered a situation contrary to our expectations. We noticed instances where the performance was worse when compared to not persisting the RDD. This unexpected outcome could potentially be

attributed to several factors. For instance, it's possible that the overhead of persisting the RDD outweighs the benefits in certain scenarios. Additionally, we speculate that due to large size of RDD disk is getting used because of the overflow which result in slower disk I/O operations hence degrading the performance



Here, we've observed the persistence of RDDs in memory-only and disk-only modes and found that there isn't much difference in the time taken between the two scenarios. This observation could be attributed to several factors:

1) **Serialization and Deserialization Overhead**: Persisting RDDs involves the serialization of data to store it in memory or disk and deserialization when it's accessed. This process incurs overhead, especially with large datasets. Both memory-only and disk-only persistence modes involve this overhead, contributing to the similarity in the time taken.
2) **Disk I/O Operations**: When persisting RDDs to disk, disk I/O operations are involved. Factors such as disk read/write speeds, disk fragmentation, and contention with other processes accessing the disk can introduce delays. However, modern disk systems often offer relatively fast read/write speeds, particularly for sequential operations, which could reduce the difference in time taken between memory-only and disk-only persistence.
3) **Minor Internal Spark Differences**: It's possible that there are minor internal differences in functionality within Spark that could contribute to the similarity in the time taken between memory-only and disk-only persistence. These differences might include

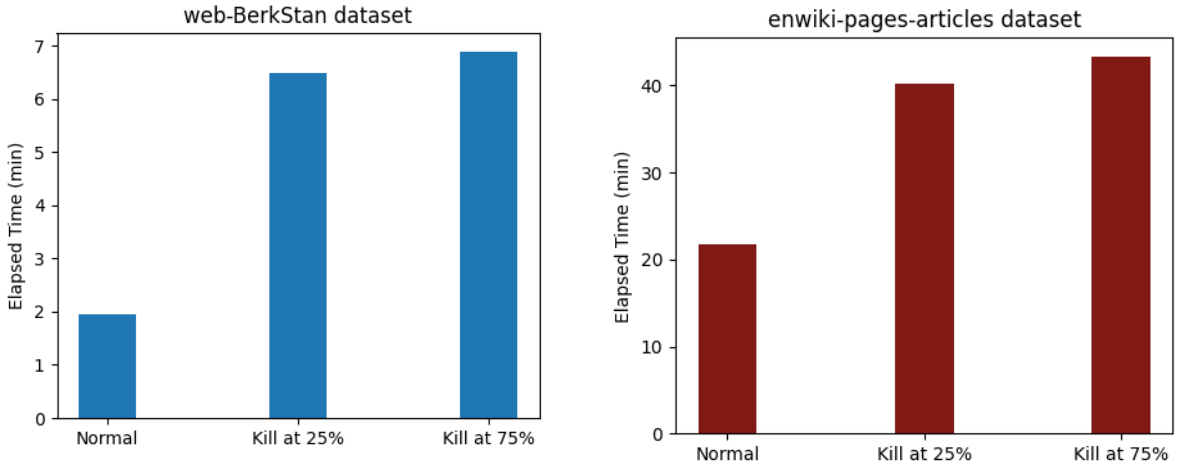
optimizations or caching mechanisms that help mitigate the performance gap between the two persistence modes.

iv Task 4: Simulated Worker Failure

We simulated worker failure scenarios in PySpark by intentionally killing a Worker process during application execution. We observed that upon killing one of the worker tasks midway through the execution of the Spark task, as expected, logs indicating lost tasks that were executing on the terminated worker. Subsequently, these tasks were re-executed on the remaining available workers. Killing a worker resulted in an expected increase in runtime, primarily due to two key reasons:

1. Lost tasks necessitated re-execution.
2. The remaining tasks had to be executed with reduced parallelism since they were distributed across two workers instead of three.

The following graph illustrates the repercussions of killing a worker.



The above figure shows the execution time on datasets with 3 scenarios-

a) Normal
b) 25% - worker process killed
c) 75% worker process killed

From the above figure we observe that if there is a failure at some node the master allocates the task to other worker using lineage but due to node failure the execution time for the task increases because of lesser number of nodes working on the task. We observe that 75% failure has little more execution time than 25% failure. This is because of wide dependencies.

Takeaways

- **Performance Analysis:** Through detailed analysis of execution times, resource utilization, and system metrics, we identified performance bottlenecks and optimized Spark applications for improved efficiency.
- **Data Distribution:** Custom partitioning strategies significantly influenced data distribution and parallelism in PySpark applications. We experimented with partitioning techniques to enhance data processing performance.
- **Fault Tolerance:** Simulated worker failures provided insights into PySpark's fault tolerance mechanisms. We observed how PySpark handles worker failures, redistributes tasks, and maintains job progress.

Contributions of Group Members

- **Yash Chaudhari –** Initial environmental setup, Part3_Task2
- **Varshita Rodda –** Hadoop Setup, Part3_Task3
- **Zhiwei Song –** Spark Setup, Part3_Task1
- **Shahab Ahmad Khan –** Part3_Task4 and graphs.
- Part 1,2 and report writeup is done by all team members.

Deliverables

We have submitted a zip file to Canvas, including:

- A report documenting our observations, findings, and task completions.
- Code for each task organized into separate folders implemented using PySpark.
- README files providing instructions on running the code.
- A run.sh script for each part of the assignment facilitating code execution on a similar CloudLab cluster.

Conclusion

Completing this assignment provided valuable hands-on experience with Apache Hadoop and Apache Spark, reinforcing our understanding of distributed data processing frameworks. Through experimentation and analysis using PySpark, we gained practical insights into system performance, fault tolerance, and optimization strategies. Overall, this assignment enhanced our skills in big data analytics and distributed computing.