Chaitanya Durgesh Nynavarapu
A20561894

ASSIGNMENT - 6

BIG DATA (CSP 554)

1) A Comprehensive Performance Analysis of Apache Hadoop and Apache Spark for Large Scale Data Sets Using HiBench

Introduction

This study investigates the performance of Apache Hadoop and Apache Spark when processing large datasets, utilizing the HiBench benchmarking suite. The researchers conducted experiments on a 10-node cluster with synthetic datasets up to 600 GB, focusing on two key workloads: WordCount and TeraSort.

Methodology

The authors examined 18 different configuration parameters related to resource utilization, input splits, and shuffle behavior. A trial-and-error approach was employed for tuning these parameters to optimize performance.

Key Findings

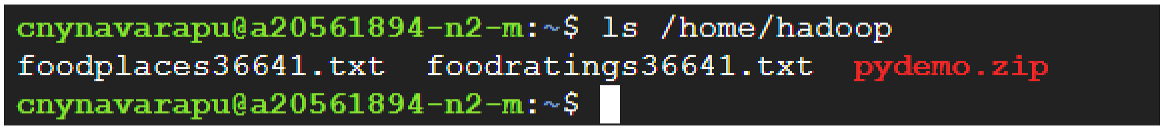
- **Performance Comparison**: Spark generally outperformed Hadoop, achieving up to **2x speedup** for WordCount and **14x speedup** for TeraSort when parameters were optimized.
- **Data Size Impact**: Both frameworks' performance heavily depended on input data size and the effectiveness of parameter tuning.
- **WordCount Efficiency**: Spark demonstrated superior execution times for WordCount, particularly for datasets up to 300 GB.
- **TeraSort Performance**: For TeraSort, Spark maintained linear performance improvements as data size increased, while Hadoop exhibited more stable throughput across varying data sizes.
- **Memory Limitations**: Spark's performance began to decline for very large input sizes (over 500 GB) due to memory constraints.
- **Parameter Sensitivity**: Adjusting input split sizes and shuffle parameters significantly influenced performance for both frameworks. Default settings were often suboptimal, especially for larger datasets.

Metrics Used

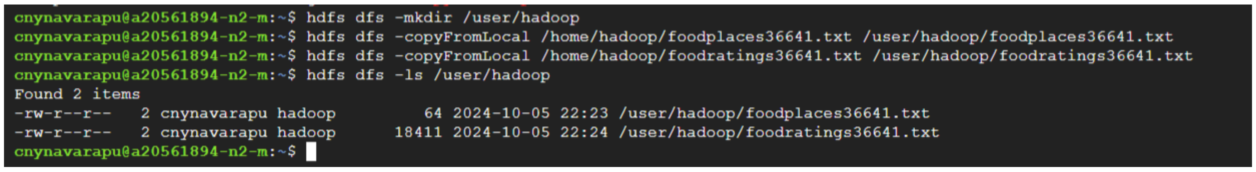
The researchers assessed performance using execution time, throughput, and speedup as key metrics. Their findings indicated that careful tuning of parameters such as input split size and shuffle buffer size could yield substantial performance improvements over default configurations.

Conclusion

While Spark generally outperformed Hadoop for smaller to medium datasets, the optimal choice between the two frameworks depends on specific workloads, data sizes, and available resources. The study underscores the importance of benchmarking and parameter tuning in big data processing frameworks to achieve optimal performance tailored to specific hardware and workload requirements.

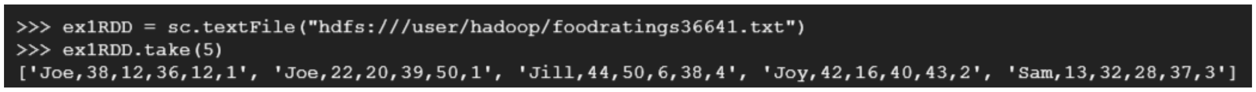


```
cnynavarapu@a20561894-n2-m:~$ ls /home/hadoop
foodplaces36641.txt  foodratings36641.txt  pydemo.zip
cnynavarapu@a20561894-n2-m:~$
```

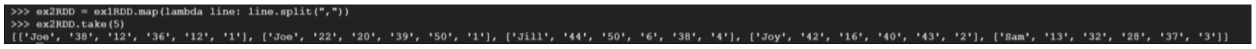


```
cnynavarapu@a20561894-n2-m:~$ hdfs dfs -mkdir /user/hadoop
cnynavarapu@a20561894-n2-m:~$ hdfs dfs -copyFromLocal /home/hadoop/foodplaces36641.txt /user/hadoop/foodplaces36641.txt
cnynavarapu@a20561894-n2-m:~$ hdfs dfs -copyFromLocal /home/hadoop/foodratings36641.txt /user/hadoop/foodratings36641.txt
cnynavarapu@a20561894-n2-m:~$ hdfs dfs -ls /user/hadoop
Found 2 items
-rw-r--r--   2 cnynavarapu hadoop         64 2024-10-05 22:23 /user/hadoop/foodplaces36641.txt
-rw-r--r--   2 cnynavarapu hadoop      18411 2024-10-05 22:24 /user/hadoop/foodratings36641.txt
cnynavarapu@a20561894-n2-m:~$
```

2) List the first five records of the RDD using the "take(5)" action and Created another RDD called ex2RDD

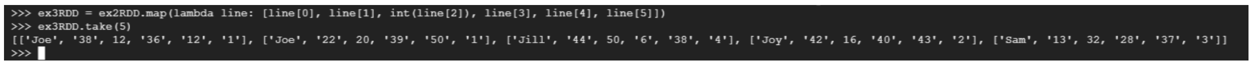


```
>>> ex1RDD = sc.textFile("hdfs:///user/hadoop/foodratings36641.txt")
>>> ex1RDD.take(5)
['Joe,38,12,36,12,1', 'Joe,22,20,39,50,1', 'Jill,44,50,6,38,4', 'Joy,42,16,40,43,2', 'Sam,13,32,28,37,3']
```



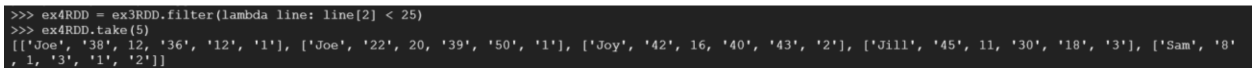
```
>>> ex2RDD = ex1RDD.map(lambda line: line.split(","))
>>> ex2RDD.take(5)
[['Joe', '38', '12', '36', '12', '1'], ['Joe', '22', '20', '39', '50', '1'], ['Jill', '44', '50', '6', '38', '4'], ['Joy', '42', '16', '40', '43', '2'], ['Sam', '13', '32', '28', '37', '3']]
```

3) Create another RDD called ex3RDD from ex2RDD where each record of this new RDD has its third column converted from a string to an integer



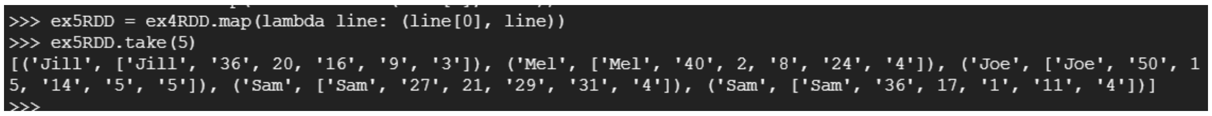
```
>>> ex3RDD = ex2RDD.map(lambda line: [line[0], line[1], int(line[2]), line[3], line[4], line[5]])
>>> ex3RDD.take(5)
[['Joe', '38', 12, '36', '12', '1'], ['Joe', '22', 20, '39', '50', '1'], ['Jill', '44', 50, '6', '38', '4'], ['Joy', '42', 16, '40', '43', '2'], ['Sam', '13', 32, '28', '37', '3']]
>>>
```

4) Create another RDD called ex4RDD from ex3RDD where each record of this new RDD is allowed to have a value for its third field that is less than 25 (<25)



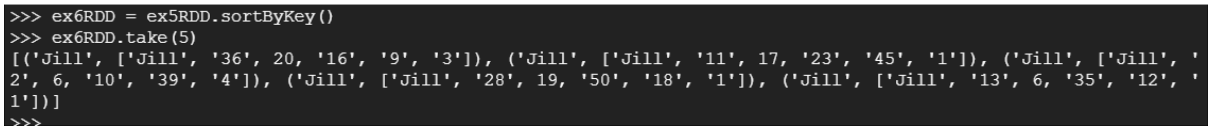
```
>>> ex4RDD = ex3RDD.filter(lambda line: line[2] < 25)
>>> ex4RDD.take(5)
[['Joe', '38', 12, '36', '12', '1'], ['Joe', '22', 20, '39', '50', '1'], ['Joy', '42', 16, '40', '43', '2'], ['Jill', '45', 11, '30', '18', '3'], ['Sam', '8'
, 1, '3', '1', '2']]
```

5) Create another RDD called ex5RDD from ex4RDD where each record is a key value pair where the key is the first field of the record and the value is the entire record



```
>>> ex5RDD = ex4RDD.map(lambda line: (line[0], line))
>>> ex5RDD.take(5)
[('Jill', ['Jill', '36', 20, '16', '9', '3']), ('Mel', ['Mel', '40', 2, '8', '24', '4']), ('Joe', ['Joe', '50', 1
5, '14', '5', '5']), ('Sam', ['Sam', '27', 21, '29', '31', '4']), ('Sam', ['Sam', '36', 17, '1', '11', '4'])]
>>>
```

6) Create another RDD called ex6RDD from ex5RDD where the records are organized in ascending order by key



```
>>> ex6RDD = ex5RDD.sortByKey()
>>> ex6RDD.take(5)
[('Jill', ['Jill', '36', 20, '16', '9', '3']), ('Jill', ['Jill', '11', 17, '23', '45', '1']), ('Jill', ['Jill', '
2', 6, '10', '39', '4']), ('Jill', ['Jill', '28', 19, '50', '18', '1']), ('Jill', ['Jill', '13', 6, '35', '12', '
1'])]
>>>
```