Big Data Processing Systems
1st Project

Helder Rodrigues
MAEBD
FCT UNL PT
Email: harr@campus.fct.unl.pt

***Abstract*—This document presents a set of experiments with the aim of showing the capabilities of 2 Big Data technologies, MapReduce and Spark RDD, for extracting insights by processing a great volume of data.**

1. Introduction

We will go trough the Project 1 statement [1] analysing the information about taxi rides in some city for better understanding the behavior of the community and help taxi drivers maximize their profit. We will use Big Data technology during our experiments, MapReduce and Spark RDD, and take conclusions about the analysis methods. We used python as programming language for all experiments.
hr
November 2, 2020

2. Dataset

The Dataset consists in a collection of taxi trip records in some city, including fields for capturing pick-up and drop-off dates/times, pick-up and drop-off locations and trip distances.

3. MapReduce

We used MapReduce to create indexes for answering the following queries:

3.1. How many trips were started in each year present in the data set?

We developed a pair of Map and Reduce programs for running on top of Hadoop.

The Mapper processes each line of the provided CSV file. It cleans its contents and extracts the respective year. As the objective is to count the trips per year and because each line represents 1 trip, it outputs the year as key *(K)* and 1 *(one trip)* as value *(V)*.

The Reducer will receive a sequence of the records extracted from the mappers, ordered by key. The objective is to sum all values for the same key and output a record with the following format: *(year,sum of trips)*. Table 1 shows an example of the obtained data.

TABLE 1. RESULTS OF 1.1

Year	Number of trips
2013	54409
2014	74753
...	...
2019	32797
2020	6829

The obtained performance counters were: real 0m1.715s / user 0m1.799s / sys 0m1.035s

3.2. For each of the 24 hours of the day, how many taxi trips there were, what was their average trip miles and trip total cost?

We developed a pair of Map and Reduce programs for running on top of Hadoop.

Here as the objective is to calculate an average, the Mapper will extract the trip miles and trip values from each line. The Reducer will sum the number of lines together with the sum of trip miles and trip cost. After iterating each key (hour) it outputs the average values by dividing the sums of trip miles and cost by the trip counts. Table 2 shows an example of the obtained data.

TABLE 2. RESULTS OF 1.2

Hour	Num trips	Avg Miles	Avg Cost
01 AM	11164	2.46	13.11
01 PM	20177	3.38	15.54
02 AM	8832	2.35	12.71
...	...	...	...
12 AM	13542	2.83	14.08
12 PM	19872	3.38	15.44

The obtained performance counters were: real 0m14.577s / user 0m16.873s / sys 0m5.236s.

TABLE 3. RESULTS OF 1.3

Hour	Route	NTrps	AMiles	ACost
01 AM	17031081700 17031081700	96	0.46	6.91
01 AM	17031081700 17031081800	73	0.54	6.96
...	...	...	...	...
08 PM	17031980000 17031980000	115	1.86	12.17
11 PM	17031839100 17031320100	64	0.69	7.08

TABLE 4. PERFORMANCE COMPARISON 1.3

Time	Default Comparator	KeyFieldBasedComparator
real	0m2.376s	0m4.661s
user	0m2.600s	0m6.075s
sys	0m1.298s	0m2.320s

3.3. For each of the 24 hours of the day, which are the 5 most popular routes according to the the total number of taxi trips? Also report and the average fare.

This problem wasn't trivially solvable with a single pair of MapReduce. Thus we created 2 MapReduce pairs: The first one behaves like the previous exercise, just adding the route to the grouping key. The second Mapper calculates a reverse order ranking, by subtracting the trip count from a very large number and concatenates it at the end of the key. The infrastructure sorts then the records by key (hour / route / order) and passes them to the second Reducer that simply prints out the first 5 records for each key. Table 3 shows an example of the obtained data.

Conversely we could opt by outputting the original counts from the Mapper and instantiate a specific Comparator using dependency injection in the infrastructure. However after testing this approach, by using *KeyFieldBasedComparator* class we realised that the performance of the second MapReduce was substantially penalised as by Table 4. We should also consider that the first MapRedude for both alternatives presented the following performance indicators: real 0m12.625s / user 0m15.753s / sys 0m5.100s.

4. PySpark

We used PySpark RDD to create indexes for answering the following queries:

4.1. What is the accumulated number of taxi trips per month?

We developed a small Python programs that instantiates the Spark context for obtaining this insight.

It starts by reading the dataset, followed by a filter for eliminating empty lines. This out-of-the-box filter function is a big simplification when comparing with MapReduce, where we would need to create Mapper and a Reducer to accomplish the same result.

TABLE 5. RESULTS OF 2.1

Month	Number of trips
7	32141
3	35260
...	...
4	32884
9	31466

TABLE 6. RESULTS OF 2.2

Pickup region	Dropoff regions
17031770202	['17031770202']
17031020301	['17031020802', '17031020301']
17031837100	['17031243000', '17031320400', '17031837100']
17031030104	['17031030104', '17031980000', '17031280100']
17031040300	['17031320400', '17031040300']
...	...

The next 2 steps *map* and *reduceByKey* behave mostly like a MapReduce pair. For the *reduceByKey* we use an operation *add* that receives 2 values and calculates the addition. Table 5 shows an example of the obtained data.

The obtained performance counters were: CPU times: user 26.3 ms, sys: 17.8 ms, total: 44.1 ms

4.2. For each pickup region, report the list of unique dropoff regions?

We had here the opportunity to use another out-of-the-box function: *distinct*. In a single line we can avoid to code a MapReduce pair of programs.

Another interesting construction was the *groupByKey* function that allows us to collect all the RDD values in a single list without the need for coding any specific iterator. Table 6 shows an example of the obtained data.

The obtained performance counters were: CPU times: user 125 ms, sys: 65.5 ms, total: 190 ms

4.3. What is the expected duration and distance of a taxi ride, given the pickup region ID, the weekday (0=Monday, 6=Sunday) and time in format "hour AM/PM"?

We used for this exercise the *aggregateByKey* function. It receives 2 parameters: The Zero value that we used to accumulate each record, the Sequential Operation function that computes the operation over two records and the Combiner Operation function that operates on the result of two Sequential Operations. In our case we used the same logic of summing all tuple members for both operations. Table 7 shows an example of the obtained data.

The obtained performance counters were: CPU times: user 2.74 s, sys: 1.11 s, total: 3.85 s

TABLE 7. RESULTS OF 2.3

Pickup_Weekday_Time	Avg Duration	Avg Distance
17031081000_6_09PM	537.40	1.10
17031838200_4_09AM	425.33	1.63
...	...	...
17031062600_5_08PM	1057.25	2.33
17031062100_3_09AM	956.00	6.90

5. Conclusion

5.1. MapReduce

- Mappers and Reducers are different standalone programs reading from Standard Input and writing in Standard Output. They don't have any dependency on any kind of Hadoop library.
- The MapReduce execution parameters can be modified by dependency injection and inversion of control on the *hadoop* command. It allows to modify program parameters like sorting comparing criteria or columns.

5.2. PySpark

- PySpark RDD has a good set of out-of-the-box functions for helping us massaging the data in a performant and scalable way without the need of writing custom MapReduce implementations. The code becomes more readable and the implementation is highly scalable by nature.
- PySpark programs are executed in a Lazy fashion. The execution graph is only triggered when we need to persist or show results. In our examples it is the execution of the *collect* function that triggers the dataflow.
- Debugging PySpark can be tricky as the execution is distributed and parallelized. One good way of accomplishing it is by using design by contract patterns like the invariants, that we can evaluate by using the *assert* function.

5.3. Performance

- We compared the performance of Exercise 1 for MapReduce and Exercise 1 for PySpark and we obtained the values presented in Table 8. We choose these 2 problems because they present similar complexity over the same dataset. We noted a performance 64 times superior when using PySpark.
- For both MapReduce and PySpark we should avoid to use lists of Python, NumPy or Pandas. Those programming patterns aren't adapted to run on top of a distributed cluster and will not be scalable.

TABLE 8. PERFORMANCE COMPARISON MAPREDUCE / PYSPARK

Time	MapReduce Ex. 1	PySpark Ex. 1
user	0m1.799s	26.3 ms
sys	0m1.035s	17.8 ms

References

[1] J. Lourenço, *Big Data Processing Systems — Project nº 1*, 2020/21. DI, FCT, UNL, PT.

[2] J. Lourenço, *Big Data Processing Systems — Class Lectures*, 2020/21. DI, FCT, UNL, PT.

[3] Thomas Erl, Wajid Khattak and Paul Buhler *Big Data Fundamentals*, 2016 Prentice Hall.