Meryem JAAIDAN,
Xavier DURAND

INF583

This report accompanies the inf583 project: database management.
In a first part, we will talk about the file organisation of the project.
Then we will talk about the project part by part and discuss the choices made.
The objective of this project is to raise awareness of the use of tools such as Spark or Hadoop to make calculations requiring important resources (memory and computing power).

CONTENTS

1 FILES ORGANISATION

You can read about this in detail in the readme.
Files are organised as a maven project (pom.xml).

The project is organised as follow :

- 'src' folder contains all sources files (the package 'A' contains the code for the part A, 'B' for the part B, and 'MatrixMultiplication' for the third question of the part B).
- 'graph' contains data of graph
- 'integers' contains data of integers

To run the part A, we just have to run the main in the Main class. As we cannot create two JavaSparkContext, you have to comment one of the two lines to test part A.

To run the part B, you just have to go to the right file and run the main function.

2 PART A

As a first step, we chose to implement the 4 questions using Spark. The choice was motivated by the functional use of the interface. You can execute the result in the Main file by running the function "sparkPartA" .

We observe that :

- the largest number is 100
- the average of all integers is 51.067
- the set of integer is those who are between 1 and 100
- there is 100 distinct integers

Then, we implement the same question using StreamingSpark. You can execute the result in the Main file by running the function "streamingSparkPartA" .
We simulate the streaming to send 100 integers per second. Furthermore, the streamingSpark reads on a window of 10 seconds, thus we can observe the modification of the result as we go along the time.
Since the beginning, we obtain the largest number of 100. Pour the average integer, we obtain it only at the end of the window, however the result at each time is a good approximation

(around 0.5).

Finally, the number of unique word is determined with the Flajolet–Martin algorithm. This algorithm can only produce a approximation. The best result is 165 or 82, it depends on the hash function (you can test two functions just by uncomment a line at the countIntegers function in the StreamingSpark class in package A). We can observe that using these hashing functions, we have a uniformely repartition of the number.
However, with this strategy, we get an unknown error at the end of the program that we don't know why it is throwing :

```
Ignoring error java.util.concurrent.RejectedExecutionException
```

Some times the error appears and others it disappears...

3 PART B

3.1 EigenVector Centrality

3.1.1 • Spark

To use this method, we first mapped each node to its neighbors to have an RDD of tuples with the edges of the graph. We also created a presentation of r_0 with RDD tuples, it is composed of the node id and its value in the vector. We join the 2 RDDs and then we have we get an RDD containing the source node and the value of each of the corresponding destination nodes id. After, we apply a reduce operation by summing the values of each source node, before we normalize the vector. To calculate the eigenvector, we repeat the process several times (30 times in our case).

3.1.2 • Hadoop

We used a two map reduce phase with the second one executed several times. The first map-reduce phase is composed of two maps. The second one is executed many times to compute the eigenvector centrality. For this second phase, we used ResultMapper, ResultReduucer and MultReducer.

3.1.3 • Threads

We implemented the eigenvector centrality usinf 5 threads. We divided the file into 5 equal parts.The EigenVectorCentrality class crreates the initial vector r_0 of type vector that is thread-safe. At each iteration we create five threads. Each one is given an id and the shared vector.

We normalize the vector at each step. The lines of the different parts will be read in a loop by each thread in order to compute the result of a row of the vector.

3.2 Most important page in Wikipedia

According to our algorithm, the most important page in Wikipedia is *Category : Rivers in Romania*.

3.3 Matrix Multiplication

For this algorithm we use the eigenvector centrality with one iteration. For the mapping phase we have two matrix A and B. For the reduce ohase, we generate 2 sorted lists for each key. We then compute the sum of the product of $A_{i,j} * B_{j,k}$. For the 2 MapReduce implementation, we followed the same steps as for the computation of eigenvector centrality using Apach Hadoop. We start by the first map phase, then the reduce phase, the second mapper phase, and finally the second reducer phase. We can finish by normalizing the resulting eigenvector values through an additional mapreduce job.

3.4 Computational cost

- Using the threads method, the computation cost will be $O(m*k+n)$ with m the number of edges, k represents the number of iterations and n represents the number of nodes.
- For the MapReduce, the cost is:
  - for the first Mapper cost = $O(m+n)$.
  - for the first Reducer cost = cost to regroup the elements with the same key together (it can be $O(log(n))$ if we are using B-tree or it can be $O(log(\alpha))$ if we are using hash map) + $o(m)$.
  - for second Mapper cost = $O(m)$.
  - for second Reducer cost = cost to regroup the elements with the same key together + $O(m)$.

  Therefore, the final cost of using one map reduce is O(m)+ cost to regroup the elements with the same key together.
- Using one MapReduce, the computation cost is :
  - for the mapper cost = $O(n^2+m)$.

- for the reducer cost = $O(n^2)$ + cost to regroup the elements with the same key together.

Therefore, the overall cost of using one map reduce is O(n2) + cost to regroup the elements with the same key together.

The results show that the performance of TwoMapReduce algorithm is better than OneMapReduce algorithm.

CONCLUSION

To put all this report in a nutshell, we implement some easy task that could use a lot of resources.
In part B, we have highlighted the so-called classical (thread) techniques for resource-intensive calculations. The importance of resource managers such as Spark or Hadoop is well noted. This was very interesting in the use of many kind of concept on Database management. It taught us how to apply our knowledge to an interesting topic, thus highlighting the performance of database algorithm.
It could have been interesting to think about a project where the time issue really arises: for example, calculating the pagerank to return google results among a huge database. However, this project was a good introduction to these issues.