INF583: SYSTEMS FOR BIG DATA

Final Project

20 mars 2022

By: Armel Randy ZEBAZE DONGMO
Paul FOTSO KAPTUE

SUMMARY

1 PRESENTATION OF THE PROBLEM

The purpose of this project was to familiarize with the use of the MapReduce paradigm to solve several Big Data problems through technologies like **Hadoop** and **Spark**. First of all, we worked on the manipulation of files of Integers with the *Java Spark API*. Furthermore, we implemented a ranking of Wikipedia Web pages based on the computation of the eigenvector centrality. In order to do that, we designed two approaches of the matrix multiplication and we compare each of them in terms of complexity and performance.

2 INTEGER MANIPULATION

In this section, the data at our disposal was a file *integers.txt* containing one integer by row. We designed MapReduce algorithms with *Apache Spark* and *Spark Streaming* to solve 4 questions.

2.1 Integer Manipulation with Apache Spark

Here, we primarily relied on the use of Resilient Distributed Datasets (RDD).

2.1.1 • Find the largest integer

The determination of the largest integer was done with one MapReduce step. We start by reading the file and mapping each of its row to integer it contained. The result of this step is a *JavaRDD*. The reduce aimed at computing the maximum. The principle is divide the RDD in chunk and successively compute the maximum. It is based on the fact that given a sequence of integers $(x_1, \ldots x_n)$ and a partition $I_1, \ldots I_k$ of $[1..n]$ we have :

$$max(x_1, \ldots x_n) = max(max\{x_{i,i \in I_1}\} \ldots max\{x_{i,i \in I_k}\}) \tag{1}$$

We obtained that the largest integer is 100.

2.1.2 • Find the average of all the integers

As for the first question, we store the file in a *JavaRDD*. We know that for a sequence $(x_1 \ldots x_n)$ of integers

$$avg(x_1 \ldots x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2}$$

We reduce the problem into the computation of the sum of all the integers because the number of integers in a RDD can be obtained in *Java Spark* with the method *count*(). The computation of the sum required one reduce step since

$$\sum_{j=1}^{n} x_j = \sum_{p=1}^{k} \sum_{j \in I_k} x_j \tag{3}$$

We obtained that the average of all the integers is 51.

2.1.3 • Output all the unique integers

We did this question with one MapReduce step. The idea was to be able to have a (key, value) data structure where the key is the integer, and the value is the number of times that integer occurs in the file. We started by using *MaptoPair()*, we mapped each integer of the RDD to a tuple $(x \rightarrow newTuple <> (x, 1))$. We obtained a *JavaPairRDD*. After that, we used *ReduceByKey()* to some the values for the same key. In order to have all the distinct integers, we kept the keys of the pairRDD and store them in an output file.

2.1.4 • Find the number of distinct integers.

This question was the simple to solve when the previous was already done. We used the previous pairRDD to which we applied the $count()$ method.
We obtained that there are 100 distinct integers.

2.2 Integer Manipulation with Spark Streaming

In this section, we relied on **JavaDStream** in order to perform the same tasks as above. This data structure works almost like a collection of *Java RDD* and propose most of their operations.

2.2.1 • Find the largest integer

We started by reading the file and storing it in a *JavaDStream<Integer>* (mapping each row to the integer it contains) that we call here the input Dstream. We applied a reduce step with $(x, y) \rightarrow max(x, y)$. The result of the reduce was a *JavaDStream* containing RDDs of one element each (the maximum element in a window/partition). We created a list to store all the maxima that we already encountered. After that, we used the method *foreachRDD()*. For each new maximum (which is actually the maximum of integers in a certain batch), we appended it to our list and output the maximum of that list as the current maximum of all our integers.

2.2.2 • Find the average of all the integers

We used a similar reasoning as with the previous question. We went from the input Dstream. We wanted to be able to have for a given batch, the sum of all its integers and their number. We started by using $mapToPair()$ with $x \rightarrow newTuple2 <> (0, newTuple <> (x, 1))$. (At this point we just had one key which is 0). We performed a $reduceByKey()$ with $(x, y) \rightarrow newTuple2 <> (x.\_1 + y.\_1, x.\_2 + y.\_2)$ and used a map to get rid of the key. At the end, and for a given batch, we had a tuple where the first component is the sum of its integers, and the second is their number. We once again used $foreachRDD()$ with 2 lists (one for the sum per batch and another one for the number of occurence per batch) to output the average.

2.2.3 • Find the number of distinct integers

We used the *Flajolet-Martin Algorithm*. It is used to approximate the number of unique elements in a data stream $X = (x_1 \dots x_n)$. The algorithm use a hash function h that maps each integer to a string of at least $\lceil log_2 n \rceil$ (the binary representation of $h(x)$). Given that representation, we compute the number of *trailing zeros* r. We have

$$r(0\dots0) = 0 \tag{4}$$

if the string is not null, r is equal to the number of consecutive zeros starting from the end. The number of distinct number is $2^{maximum\ number\ of\ trailing\ zeros}$.
The only thing we had to do was to choose a hash function (for example $h : x \to h(x) = (3 * x + 5)\%(2^7)$) and compute the number of trailing zeros of $h(x)$ for each integer x. With a *JavaDStream* containing that (number of trailings zeros) we just had to apply the same reasoning as 2.2.1 and print 2^{max} rather than max.

3 RANKING WIKIPEDIA WEB PAGES

In this section we received a graph G where nodes represent Wikipedia Web pages and are stored in the file *idslabels.txt*, the edges are the hyperlinks between theses pages and are stored in the file *edgelist.txt*. A line in *idslabels.txt* contains an index followed by the name of a page. Each line of *edgelist.txt* consists of the index of the page followed by the indices of the pages linked to it. there are $n = 64375$ nodes and $m = 818535$ edges. We used the API PySpark to implement all our algorithms.

3.1 Eigenvector centrality implementation

3.1.1 • With Apache Spark

For the computation of the eigenvector centrality in Apache Spark, we relied on a 2-steps matrix multiplication that we're going to explain in detail later on. The computation of the norm of the vector is done via one MapReduce, a square mapping $(x \to x^2)$ and a reduce with $(x, y) \to x + y$.

3.1.2 • With Hadoop

3.1.3 • With the Threads

3.2 Most important page in Wikipedia

In order to obtain the most important page, we used *idslabels.txt*. As a matter of fact, we used that file to build a *PairRDD<Integer, String>* that maps each page to an index (*nodes*). When we obtained a representation of the vector $r_{t_{final}}$ as a pairRDD<Integer, Double> (index -> importance), we used the method $max()$ to get the index with the maximum importance value. We applied the method $lookup()$ to find the corresponding name in the *Nodes* RDD.

3.3 Matrix Multiplication implementation

We can define an element of a matrix by three attributes : the row number, the column number, and the corresponding value. That's what tipped us off on how to store the different matrices in RDDs.

We stored the two matrices in two different RDDs in which each line respectively has the form (i, j, a_{ij}) and (j, k, b_{jk}) we don't store null elements, that is very efficient in terms of space for sparse matrices.

3.3.1 • Using two MapReduce steps

— We used a first Map to send each matrix element a_{ij} to the (key, value) pair : $(j, (A, i, a_{ij}))$
— Simultaneously we send each matrix element b_{jk} to the (key, value) pair : $(j, (B, k, b_{jk}))$
— For each key j, we examined its list of associated values. For each value that comes from A, say (A, i, a_{ij}), and each value that comes from B, say (B, k, b_{jk}) we produced the tuple $(i, k, val = a_{ij}b_{jk})$
— We used a reduce operation to output a key j paired with the list of the tuples of the previous form associated to j : $(j, [(i_1, k_1, val_1), (i_2, k_2, val_2), \dots, (i_q, k_q, val_q)])$
— From the previous pair we produced q (key, value) pairs :

$$((i_1, k_1), val_1), ((i_2, k_2), val_2), \dots, ((i_q, k_q), val_q)) \tag{5}$$

— For each key (i, k), we used a reduce operation to sum all the values associated with this key and we ended up with a pair $((i, k), v)$ where v is the value of the element in row i and column k of the product.

3.3.2 • Using one MapReduce step

— We used one map in order to, for each element a_{ij} of A, produce a (key, value) pair $((i, k), (A, j, a_{ij}))$, for $k = 1, 2, \dots$ up to the number of columns of B.
— Simultaneously, for each element b_{jk} of B, produce a key-value pair $((i, k), ((B, j, B_{jk}))$ for $i = 1, 2, \dots$ up to the number of columns of A.
— For each key (i, k) we have an associated list with all the values (A, j, a_{ij}) and (B, j, b_{jk}) for all possible values of j.

- We associated the two values on the list that have the same value of j, for each j :
  - We sort by j the values that begin with A and sort by j the values that begin with B, in separate lists.
  - The j^{th} value on each list must have their third components, a_{ij} and b_{jk} extracted and multiplied.
  - Finally, these products are summed up and the result is paired with (i, k) by a reduce operation.

3.4 Cost of computing the eigenvector centrality and influence of the number of MapReduce steps

Because the computation of the total cost uses the cost of the matrix multiplication, we will distinguish two cases : the computation with two MapReduce steps and the computation with one MapReduce step. We have the following formula :

$$cost = \#iterations \times (cost(matrix\ multiplication) + cost(vector\ normalization)) \tag{6}$$

We won't care about the cost of the operations involved in the map and the reduce. We will rather focus on the amount of tuples transferred from the result of a mapping to the application of a Reduce.

Note : The most important step when it comes to normalize a vector is the computation of its norm. That can be done with one MapReduce step as we explained earlier, a square mapping followed by a Reduce with $(x, y) \to x + y$. The number of tuples exchanged in this operation is approximately $\mathcal{O}(size\ of\ the\ vector)$.

3.4.1 • Cost with two MapReduce steps

- The first Map produces m tuples coming from A because we only stored non-zero's elements and (at most) n tuples coming from B. We end up sending $m + n$ tuples from the first map to the first Reduce operation.
- The second Map produces for each j at most $degree(j)$ tuples, we do that for all j and we end up with $m = (\sum_{1 \leq i,j \leq n} A_{ij} 1_{A_{ij} \neq 0})$ tuples which are sent to the second Reduce operation.
- With the help of the formula above, we obtain that the total cost of the algorithm is :

$$cost = k \times ((2m + n) + \mathcal{O}(n)) \tag{7}$$

3.4.2 • Cost with one MapReduce step

- The Map operation produces m tuples coming from A and n^2 tuples coming from B so we ended up sending $m + n^2$ tuples from the Map to the Reduce operation.
- With always the help of the formula above, we obtain that the total cost of the algorithm is :

$$cost = k \times ((m + n^2) + \mathcal{O}(n)) \tag{8}$$

RÉFÉRENCES