INF583 Project

Michael Fotso Fotso, Tristan Francois

michael.fotso-fotso@polytechnique.edu, tristan.francois@polytechnique.edu

March 20, 2022

1 Introduction

2 Lists of integers

All the algorithms of this exercise have been implemented with spark.

2.1 The largest integer

This algorithm is very simple. In the mapping phase, we simply return all the integers with the same key. In the reduce phase, we keep track of the largest integer we've seen for the single key and we return it at the end.

Input: List of integer
Output: The largest integer in the list
Map:
  foreach $i \in value$ **do**
    emit pair(1, i)
  end
Reduce:
  $largest \leftarrow -\infty$
  foreach $i \in value$ **do**
    $largest \leftarrow \max(largest, i)$
  end
  emit pair(key, $largest$)

Algorithm 1: MapReduce algorithm to produce the largest integer

We found that the largest integer in the list was 100.

2.2 The average of all the integers

Here, the mapping phase is the same as in the previous question, i.e. we return all integers with the same key. For the reduction we sum all the integers and count the total number of integers. Then we return the sum divided by the number of integers.

Input: List of integer
Output: The average of all the integers
Map:
  foreach $i \in value$ **do**
    emit pair(1, i)
  end
Reduce:
  $sum \leftarrow 0$
  $n \leftarrow 0$
  foreach $i \in value$ **do**
    $sum \leftarrow sum + i$
    $n \leftarrow n + 1$
  end
  emit pair(key, $\frac{sum}{n}$)

Algorithm 2: MapReduce algorithm to produce the average of all the integers

We found that the average of the list was 51.067.

2.3 The same set of integers, but with each integer appearing only once

This time, the mapping consists in returning the integer as key, followed by any value. For the reduction, we just return the key.

Input: List of integer
Output: The same set of integers, but with each integer appearing only once
Map:
  foreach $i \in value$ **do**
    emit pair(i, 0)
  end
Reduce:
  emit pair(0, key)

Algorithm 3: MapReduce algorithm to produce he same set of integers, but with each integer appearing only once

2.4 The count of the number of distinct integers in the input

To do this, we first pass the list of integers into the algorithm 3. Then we pass the result - the list of distinct integers - in the algorithm below.

Here, the mapping consists in returning only pairs (1,1). Since we take as parameter a set of unique integers, we obtain as many couples as there are integers in the set. Then we just have to count the number of pairs in the reduction phase.

We found that there were 100 distinct integers in the list.

Input: List of distinct integer
Output: The count of the number of distinct integers in the input

```
Map:
    foreach i ∈ value do
        emit pair(1, 1)
    end
Reduce:
    count ← 0
    foreach i ∈ value do
        count ← count + i
    end
    emit pair(key, count)
```

Algorithm 4: MapReduce algorithm to produce the count of the number of distinct integers in the input

2.5 The largest integer in Spark streaming

To do this, we started by creating a local QueueStreaming data with batch size 15 to be able to test our algorithm. We also set a maximum streaming time (30 seconds). The general method consisted in creating a static global variable intended to contain the maximum. Each time a data stream is received, we compare the maximum value of the data stream to our global variable and we update if necessary. Thus, at each time t, the global variable contains the maximum value of all received data.

Input: DataStream of list of integer
Output: The largest integer in the list

```
Global_largest ← −∞
foreach stream ∈ DataStream do
    Map:
        foreach i ∈ stream_value do
            emit pair(1, i)
        end
    Reduce:
        largest ← −∞
        foreach i ∈ value do
            largest ← max(largest, i)
        end
        emit pair(key, largest)
    Global_largest = max(Glabal_largest, largest)
end
```

Algorithm 5: MapReduce algorithm to produce the largest integer in spark streaming

2.6 The average of all the integers in spark streaming

The principle here is very similar to that of the previous part. We start by creating a global static variable (a, b) with a representing the sum of all elements encountered so far and b the total number of elements already encountered. The values of a and b are updated as the streams arrive.

Input: DataStream of list of integer
Output: The average of all the integers
$(Global\_sum, Global\_count) \leftarrow -(0, 0)$
$Global\_mean = 0$
foreach $stream \in DataStream$ **do**
  Map:
    foreach $i \in stream\_value$ **do**
      emit pair(1, i)
    end
  Reduce:
    $sum \leftarrow 0$
    $count \leftarrow 0$
    foreach $i \in value$ **do**
      $sum \leftarrow sum + i$
      $count \leftarrow count + 1$
    end
    emit pair(key, $(sum, count)$)
  $(Global\_sum, Global\_mean) = (Global\_sum + sum, Global\_count + count)$
  $Global\_mean = Global\_sum/Global\_mean$
end

Algorithm 6: MapReduce algorithm to produce the average of integers in spark streaming

2.7 The count the number of distinct integers in the input in spark streaming

Here, we used the Flajolet-Martin algorithm to approximate the number of distinct elements after a streaming period. As in the previous two parts, we used a global static variable that stores the largest r (the number of zeros at the end of a number) of the Flajolet-Martin algorithm and predicted the number of distinct elements as 2^r. We used the identity as a hash function in the Flajolet-Martin algorithm.

Input: DataStream of list of integer
Output: The count of the number of distinct integers

```
1  Global_r ← 0
2  Global_count ← 0
3  foreach stream ∈ DataStream do
4  |  Map:
5  |  |  foreach i ∈ stream_value do
6  |  |  |  temp ← i%2
7  |  |  |  temp ← base2(temp)
8  |  |  |  temp ← trailingZeros(temp)
9  |  |  |  emit pair(key, temp)
10 |  |  end
11 |  Reduce:
12 |  |  r ← 0
13 |  |  foreach i ∈ value do
14 |  |  |  r ← max(r, i)
15 |  |  end
16 |  |  emit pair(key,r)
17 |  Global_r ← max(Global_r, r)
18 |  Global_count ← 2^Global_r
19 end
```

Algorithm 7: MapReduce algorithm to count the number of distinct integers in spark streaming

3 Ranking Wikipedia Web pages with a centrality measure

3.1 Question 1

3.1.1 Spark

With spark, we first transform A into JavaPairRDD (i, j) such that $A[i, j] = 1$. Then we created r_0 in JavaPairRDD (i, val) such that $val = r_0[i]$. Then we computed the matrix product by doing a join of A and $r0$ according to the columns of A and according to the indices of r_0 followed by a reduce. The rest of the algorithm is implemented by a loop of map reduces.

3.1.2 Hadoop

Here, we have used a two-step matrix product implementation of map reduce. We will describe how this matrix product works in question 3.

Just note that for each iteration, the files were stored in a folder iteration t which is used as input at iteration $t + 1$.

3.1.3 Thread

To deal with the problem by threads, we have adopted a simple method by implementing the block matrix multiplication algorithm. We have partitioned the matrix in 4 blocks, and we have created a thread in charge of performing the multiplication corresponding to its block.

The graph is stored as an adjacency set, i.e. an array of size the number of nodes of the graph, where each element i of the array is a set containing all the neighbors j of i. Using sets instead of lists allows us

to test in constant time the neighborhood membership of a node.

Our algorithm then corresponds to the naive matrix multiplication algorithm. We run the indices of the matrix corresponding to the hypothetical adjacency matrix in a double for loop. Then, we test if there is indeed an edge between the two nodes using the adjacency set, and we add the corresponding value.

Input: A the adjacency set, r the eigenvector of the pages the s_i the starting line of the block, e_i the ending line of the block, s_j the starting column of the block, e_j the ending column of the block.

Output: The product vector for the block.

```
1 res ← new double[e_i − s_i]
2 foreach i ∈ [s_i .. e_i] do
3 |   res[i − s_i] ← 0
4 |   foreach j ∈ [s_j .. e_j] do
5 |   |   if j ∈ A[i] then
6 |   |   |   res[i − s_i] ← res[i − s_i] + r[j]
7 |   end
8 end
9 return res
```

Algorithm 8: Thread for block matrix multiplication

We then obtain four result vectors that we just have to combine to complete the matrix multiplication. There is no problem of concurrent memory access, because no thread writes to a shared memory.

3.2 Performance

You will find the execution times of the different versions in the table below for T=10. Spark is clearly the fastest version, while the Thread and Hadoop versions are comparable in terms of performance.

Hadoop	Spark	Thread
57274	10031	59828

3.3 Question 2

We answered this question with Spark. We started by determining the index of the vector with the largest eigenvector centrality using the previously calculated vector. We then transformed the list of ids into JavaPairRDD $(i, name)$ such that i is the id of the page and $name$ is the name of the page. Finally we applied a filter to detect the indices that correspond to the largest eigenvector Centrality. The best page was finally:$Categorie : River$

3.4 Question 3

We used Hadoop to implement this question. Using a single step of map reduce resulted in a decrease in performance.

3.4.1 Two step

We suppose that each row of the vector starts with a character r which allows to know that the row comes from the vector rather than the matrix. The first step is to make a join between the matrix and the vector according to the columns of the matrix and the rows of the vector, for this we start with a map taking as input both the vector and the matrix. For each line, if it comes from the vector we write it in the context $(i, "r" + " " + val)$ such that $r[i] = val$. If the line comes from the matrix, we write in the context (j, i) with i the index of the line and j all the corresponding columns. In the reducer, for each key j, we will have a set of values i and only one value containing the character r from which we extract $val = r[j]$. So we write in the context $(i + " " + j, val)$. So we have made the join.

The second one is simply to make a $map(i, j, val) -> (i, val)$ then a reducer $(a, b) -> a + b$. This allows to calculate the matrix product.

3.4.2 One step

It is assumed that each row of the vector starts with a character r which allows to know that the row comes from the vector rather than the matrix. We also assume that the size of the matrix is known. In the map, if the line comes from the matrix, we write all the (i, j) in context where i is the index of the line and j the corresponding columns. If the line comes from a vector (thanks to the character r) we write in for the context $(i, "r\ " + j + "\ " + val)$ with $val = r[j]$ and for i going from 1 to n with n the number of nodes.
In the reducer, you just have to choose among the n values coming from the vector (detect thanks to the character r) those which appear in the set of j which comes from the matrix. then you calculate the sum of these values in a variable sum and you write in the context (i, sum).

3.5 Performance

The performance of the two-step multiplication is much better than the one-step version. Indeed, in our example, the two-step version took 4126 milliseconds to calculate the product Ar while the one-step version took 204114 milliseconds.

3.6 Question 4

Whether it is the version with two-step multiplication or one-step multiplication. There are three important steps whose cost must be evaluated: matrix multiplication, norm calculation and normalization.

For both versions, the calculation of the norm uses a mapping of cost n and a reduce of cost n. Thus the total cost of computing the norm is $2n$.

Similarly, for both versions the cost of normalization is $2n$.

We will now detail the cost of the multiplication for both versions.

3.6.1 Two steps

The cost of the first map is $n + n$. The cost of the first reduce is $n + m$. The cost of the second map is m and finally the cost of the second reduce is m. In total the cost of the matrix multiplication is $3n + 3m$.

The total cost of this version is $k(6n + 3m)$ where k is the number of iterations.

3.6.2 One steps

The cost of the map is $n+n$ and te cost of the reduce is n^2+m. In total the cost of the matrix multiplication is $n^2 + 2n + m$.

The total cost of this version is $k(n^2 + 5n + m)$ where k is the number of iterations.