Victoire de Termont
Sarah Jallot
Naomi Serfaty

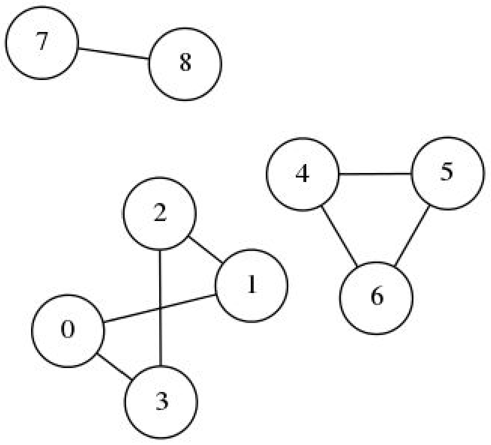
Database Management project: Finding connected components in graphs

For this database project, we decided to focus on the first option, which purpose was to find connected components in graph. The goal of this paper is to transmit what we have understood from the problem, how we managed to build an efficient algorithm and implement it locally and then on a big scale using RosettaHub.

I. Description of the adapted solution

1. Understanding what connected components are

To start with, our work is based on the paper from Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Sun Data Research, called *CCF: Fast and scalable connected component computation in MapReduce*. The paper was very useful to understand what connected components were, and how we could find the groups in a relatively easy way. Our method was to understand the principle on a small example, then understand the pseudo-code and try to find a manual manner to get the good result using the pseudo-code steps.



First, to better understand what connected components were, we decided to use a small example as the one on the left. Here, we can see that components 7 and 8 are connected, but not connected to any other point (called edge). Thus, 7 and 8 are connected components and form a group.
In the same manner, 4, 5 and 6 are another group, and 0, 1, 2, 3 the last one of that example.

Then, we understood that we needed a key to identify the group corresponding to each edge, and that a good way of choosing the key was to take the minimum value of all edges belonging to each group.

Thus, coming back to our example, finding the connected components in this graph would consist in obtaining the following result: (0,0), (1,0), (2,0), (3,0), (4,4), (5,4), (6,4), (7,7), (8,7).

Once we had understood what the goal of the algorithm was, we decided to look into details at the pseudo-code that was given, to understand the methodology. Indeed, as the goal was to rewrite the algorithm in Spark / PySpark, we had to understand it very well in order to extract the main ideas from it.

Note that the input is not a drawing of the graph but some information on connections, that can be in one way or both. For instance, with our example, we could have the following input: (7,8), (4,5), (6,5), (4,6), (6,4), (1,2), (3,2), (2,3), (0,3), (1,0), (2,1). We note each tuple (k,v).

2. Extracting the general ideas from the given pseudo-code

This main pseudo-code that we used was the following one, and here is our understanding of what we should try to implement in our algorithm. It might not follow the exact same process as the pseudo-code, but we kept the general ideas to design our algorithm:

```
CCF-Iterate
map(key, value)
   emit($key, value$)
   emit($value, key$)

reduce($key, < iterable > values$)
   $min \leftarrow key$
   for each ($value \in values$)
      if($value < min$)
         $min \leftarrow value$
      valueList.$add(value)$
   if($min < key$)
      emit($key, min$)
      for each ($value \in valueList$)
         if($min \neq value$)
            Counter.NewPair.$increment(1)$
            emit($value, min$)
```

<u>Step 1:</u> Do a Map to extract all the existing connections from the given input, for each (k,v) we want to emit (k,v) and (v,k). The goal of this first action is to ensure that we have all the connections in both ways, as it will make it easier for the next steps.

<u>Step 2:</u> Do a Reduce to group by key for each edge, with the list of all directly connected edges as value

<u>Step 3:</u> Put the smallest value in k and v as the new key, so that the temporary key becomes the minimum value of all edges connected within each tuple

<u>Step 4:</u> for each element of v, emit (v[i], k), where k is the temporary key - smallest identified connection, direct or indirect.

From the CFF-DEDUP pseudo-code and explanations, we understood that we needed to emit distinct tuples in our last step to increase efficiency of speed.

Then, the idea is to loop on the previous steps to get the final results. Indeed, looping makes it possible to get first, second, third... indirect connections, with a decreasing group key, as we extract the minimum value of the group.

Also, we noticed that the final result should output a number of tuples equal to the number of distinct edges given in the input, as we want to have each edge and the key of the correspondent group. Before reaching the final result, we automatically have a bigger number of tuples outputted, as we have more listed connections than necessary. This reasoning enables us to know when to stop the looping process.

3. Writing our own algorithms

The main goal of this project was to understand the previous algorithm, and to implement it in PySpark, using an RDD object and a Dataframe object.

We started by the RDD process, designing functions to achieve our goal:

- build a function to import a .txt file as an RDD object
- design a function to execute the steps described above at once
- build the final model, looping on our step-by-step function and outputting the result

Then, we did the same thing but importing the file as a Dataframe. The difficulty was that PySpark functions had to be used differently depending on the type of input.
Another difficulty that we encountered, and that we will elaborate on in the second part of this report, was that some functions were available in a recent version of PySpark but not in the version currently used on the RosettaHub machine, so we had to recode them manually using udf.

To ensure that both algorithms were working correctly, we verified that we were getting the same results out of both of them.

II. Designed algorithms with explanations and comments

As explained in part I of our report, we coded our algorithm using PySpark, as we prefered it to Scala. We installed many packages and imported them afterwards, but it is not very relevant so we decided not to display them in this section (the full code is available in the appendix).

1. Algorithm using RDD format

We started by defining a function, called **file_to_rdd()** to transform our txt file into an RDD, readable by PySpark.

The first remark is that this function can take as input either a .txt file or .csv file without causing any trouble. Our final dataset comes from a txt file with a specific format but for experimental purposes we created our own graphs in csv files which explains why this function accepts both types.

Note that we deleted the first 3 text lines at the beginning of the web-Google.txt file for simplicity, as it was once again not directly related to the algorithm.

```
def file_to_rdd(file):
    """
    This function takes a file name and converts it into an RDD.

    Arguments:
    file (str): file name

    Returns:
    An RDD containing all information extracted from the file.
    """

    if file[-3:] == "csv" :
        data = spark.read.format("csv").option("inferSchema", "true")\
                                        .option("delimiter", ',')\
                                        .option("header", 'true')\
                                        .load(file).cache()

        rdd = adj_cache.rdd.map(tuple)
        return rdd
    elif file[-3:] == "txt" :
        rdd_web = sc.textFile(file) \
                    .map(lambda line: line.split('\t')) \
                    .filter(lambda line: len(line)>1) \
                        .map(lambda line: (line[0],line[1]))

        return rdd_web
```

Then, we defined the **CCF_DEDUP_rdd()** function, that is at the heart of our connected components search. When inputting an RDD, the output of this function is an RDD, giving the smallest (temporary) connected component for each edge.
The function works as follows:

The input is an RDD with tuples of connected components (i.e edges). We add to the initial RDD all reversed tuples, called **rdd_reverse**, as if connections in both ways were possible for all edges, and call this obtained result **rdd_0**.
The goal of this preliminary step is to enable the extraction of the minimum value within each group in the next steps.
For instance, an RDD containing (1, 2) will make (1, 2) and (2, 1) go into rdd_0.

Then, the algorithm groups the tuples by key, and the value becomes a distinct list of all the elements that were grouped. We thus obtain for each node, a list of one or more nodes connected to the node. After grouping the tuples by key to obtain only one single list by node, we call this new RDD **rdd_1**.
For example, (1, 2), (3, 4), (3, 5), (2, 1) becomes (1, (2)), (3, (4, 5)), (2, (1)).

From rdd_1, we build a new RDD called **rdd_2**, where each tuple has a key equal to the minimum value found within the whole tuple in rdd_1, and then a value equal to the list of all elements of that tuple (former k and elements in v).
For example, (4, (5, 3, 2)) becomes (2, (5, 3, 2, 4)).

Finally, the **output rdd_3** is a list of tuples, having for key an edge and for value the minimum assigned to it previously. The aim of this step is to access distanced connections as we iterate on the function.
For example, our previous tuple (2, (5, 3, 2, 4)) will output the following tuples: (5, 2) , (3, 2), (2, 2) and (4, 2).

```python
def CCF_DEDUP_rdd(rdd):
    """
    This function takes an RDD and returns for each component the closest
    connected neighbor, in one way or the other: we can have (a,b) or (b,a), or both.
    It is inspired from the CCF_iterate and and the CCF_Dedup found
    in the article https://www.cse.unr.edu/~hkardes/pdfs/ccf.pdf

    Arguments:
    rdd (rdd): rdd name

    Returns:
    An RDD containing for each component the closest connected
    neighbor(direct relationship only),in one way or the other.
    """

    # Our goal is to list all existing edges in both ways: (k,v) and (v,k)
    # Our called RDD contains all (k,v) and we want to add all (v,k)

    rdd_reverse = rdd.map(lambda x :(x[1], x[0])) # getting all (v,k)
    rdd_0 = rdd.union(rdd_reverse) # Building a new RDD containing all (k,v) and (v,k)

    # Grouping by key on the first element (k, [v1, v2...])
    rdd_1 = rdd_0.groupByKey().map(lambda x : (x[0], list(x[1])))

    # New k: the minimum between k and all elements included in v
    # New v: all values from k and v
    rdd_2 = rdd_1.map(lambda x : (min(x[0], min(x[1])),  list(set(x[1] + [x[0]]))))

    # Extracting each element of v as our key k and assigning it the corresponding minimum found above
    rdd_3 = rdd_2.flatMapValues(lambda x : x).map(lambda x : (x[1], x[0])).distinct()

    return rdd_3
```

When looping on this function, neighbors that are one step closer than during the previous output are identified. In the end, if we loop the function enough times, we obtain an RDD containing each node as a key and the corresponding group name as the value (i.e. the id of the node with the smallest value in the group).
Thus, we built a final function, called **get_groups_rdd()** to obtain the final result looping on CCF_DEDUP_rdd() function.

As explained in part I of the report, we know that we can stop looping when we obtain an RDD with as many tuples as the number of distinct nodes (distinct values of k and v) in the original RDD. Thus, we start by counting the number of nodes to be able to tell the algorithm when to end the looping process.

The output of our function contains the computational time, the total number of distinct nodes, the number of groups of connected components, the number of times the algorithm goes through the loop (called counter), and all the distinct nodes with their assigned groups.

We decided to output the run time to compare RDD and Dataframe methods.
The total number of distinct nodes and the number of groups of connected components were outputed because we wanted to be able to understand the time variation criteria taken into account by each algorithm (is it the number of edges that makes time vary ? is it the number of nodes? is it the number of groups ? it is the distance between nodes ?).
The counter allowed to understand how many times the algorithm needed to loop on the CCF_DEDUP_rdd() to return the output.
Finally, getting all the distinct nodes and their assigned groups was the main goal of the algorithm.

```python
def get_groups_rdd(rdd):
    """
    This function extracts connected components from an RDD, and assigns
    the smallest component value of each group as the group name.

    Arguments:
    rdd (rdd): rdd name

    Returns:
    t (float) : Computational Time
    size (int) : Total number of distinct edges in the input graph
    num_of_groups (int) : Number of groups of connected components
    rdd (rdd) : An RDD containing as many tuples as the number of unique components in
    the original RDD: the key is the component and the value is the group name.
    """

    #Count of edges
    rdd_reverse = rdd.map(lambda x :(x[1], x[0])) # getting all (v,k)
    rdd_0 = rdd.union(rdd_reverse) # Building a new RDD containing all (k,v) and (v,k)
    size = (rdd_0.distinct()).count()/2

    # Final number of tuples must be equal to the number of distinct values in our original RDD
    # And we can prove that they are equal only once the solution is found
    t = time.time()

    #The counter counts the number of times the algorithm goes through the loop
    #It will depend on the maximum distance between two components of the same group
    counter = 0
    while rdd.count()!= (rdd.groupBy(lambda x : x[0]).distinct()).count() :
        counter +=1
        rdd = CCF_DEDUP_rdd(rdd) # function explained above
    t = time.time() - t

    #Getting the number of groups of connected components
    num_of_groups = len(rdd.values().distinct().collect())

    return t, size, num_of_groups, counter, rdd
```

Please note that we decided not to repartition our rdd as the program was running fast enough and didn’t need more parallelization.

2. Algorithm using Dataframe format

To write the algorithm using a Dataframe input, we decided to use our algorithm taking an RDD as input and to adapt it to the Dataframe format.

We started by defining a **file_to_df()** function to encode our txt file into a Dataframe readable by PySpark. Nothing particular has to be specified, we simply adapted the format and gave names to our columns, as it was done in the original txt file.

```
def file_to_df(file):
    """
    This function takes a file name and converts it into a DataFrame.

    Arguments:
    file (str): file name

    Returns:
    An DataFrame containing all information extracted from the file.
    """

    if file[-3:] == "csv" :
        data = spark.read.format("csv").option("inferSchema", "true")\
                                    .option("delimiter", ',')\
                                    .option("header", 'true')\
                                    .load(file).toDF("To","From").cache()
        adj_cache = data.persist()

        return adj_cache
    elif file[-3:] == "txt" :
        df_web = sc.textFile(file) \
            .map(lambda line: (line.split('\t'))).toDF()\
            .select(col('_1').cast(IntegerType()).alias('To'), col('_2').cast(IntegerType()).alias('From'))

        #Remove header if there
        #df_web = df_web.filter(df_web.To !='FromNodeId' )

        return df_web
```

Then, we coded our **CCF_DEDUP_df()** function, which goal was to output exactly the same thing as the CFF_DEDUP_rdd() function but with a Dataframe format.
We started by using PySpark predefined functions such as array_union, array_min and array_distinct - that are new (version 2.4). They were working very well locally. However, when we put our code on RosettaHub, those functions were not recognized, maybe because the version of spark on RosettaHub was not up to date.
Thus, we decided to work around the problem by creating UDFs (user defined functions), that allowed us to code a Python function into a PySpark one. We called them our_union_udf, our_min_udf and our_distinct_udf.

```
def our_union (x):
    x[1].append(x[0])
    return x[1]

our_union_udf = f.udf(our_union, ArrayType(IntegerType()))
our_min_udf = f.udf(lambda x: min(x), IntegerType())
our_distinct_udf = f.udf(lambda x: list(set(x)), ArrayType(IntegerType()))
```

The **CCF_DEDUP_df()** function outputs the same thing as the **CCF_DEDUP_rdd()** function, but with all tuples put into a Dataframe format instead of an RDD. However, as available functions are different in PySpark for RDDs and Dataframes, we had to write the code in a different way.
Each step does exactly the same thing as in the **CCF_DEDUP_rdd()** function, that we detailed in part II, 1.

```python
def CCF_DEDUP_df(df):
    """
    This function takes an DataFrame and returns for each component the closest
    connected neighbor, in one way or the other: we can have (a,b) or (b,a), or both.
    It is inspired from the CCF_iterate and and the CCF_Dedup found
    in the article https://www.cse.unr.edu/~hkardes/pdfs/ccf.pdf

    Arguments:
    df (df): DataFrame name

    Returns:
    An DataFrame containing for each component the closest connected
    neighbor(direct relationship only),in one way or the other.
    """

    # Our goal is to list all existing edges in both ways: (k,v) and (v,k)
    # Our called RDD contains all (k,v) and we want to add all (v,k)
    reverseDF = df.select(col("From").alias("To"),col("To").alias("From"))# getting all (v,k)
    df_0 = df.union(reverseDF)# Building a new DataFrame containing all (k,v) and (v,k)

    # Grouping by key on the first element (k, [v1, v2...])
    df_1 = df_0.groupBy(col("To")).agg(our_distinct(collect_list(col("From"))).alias('From'))

    # New k: the minimum between k and all elements included in v
    # New v: all values from k and v
    df_2 = df_1.withColumn('From', our_union_udf(struct(df_1.To, df_1.From)))\
                .withColumn('To', findmin("From"))\
                    .withColumn('From', our_distinct('From'))

    # Extracting each element of v as our key k and assigning it the corresponding minimum found above
    df_3 = df_2.select( explode(col("From")).alias("To"), col("To").alias("From")).dropDuplicates()

    return df_3
```

The final function **get_groups_df()** does the exact same thing as **get_groups_rdd()**. Once again, the only difference is the format of the last output.

```python
def get_groups_df(df):
    """
    This function extracts connected components from a DataFrame, and assigns
    the smallest component value of each group as the group name.

    Arguments:
    df (df): DataFrame name

    Returns:
    t (float) : Computational Time
    size (int) : Total number of distinct edges in the input graph
    num_of_groups (int) : Number of groups of connected components
    df (DataFrame) : An DataFrame containing as many tuples as the number of unique components in
    the original DataFrame: the key is the component and the value is the group name.
    """

    #Count of edges
    reverseDF = df.select(col("From").alias("To"),col("To").alias("From")) # getting all (v,k)
    df_0 = df.union(reverseDF)# Building a new DataFrame containing all (k,v) and (v,k)
    size = df_0.distinct().count()/2

    # Final number of tuples must be equal to the number of distinct values in our original RDD
    # And we can prove that they are equal only once the solution is found
    t = time.time()

    #Counter measures the max distance between two neighboors of the group
    counter = 0
    while df.count()!= df.select('To').distinct().count() :
        counter +=1
        df = CCF_DEDUP_df(df) # function explained above
    t = time.time() - t

    #Getting the number of groups of connected components
    num_of_groups = len(df.select('From').distinct().collect())

    return t, size,num_of_groups, counter,  df
```

Note that we didn't check that our algorithms were working correctly on our local machine with the whole target document (web-Google.txt), but with small examples that we designed ourselves, to have access to the results we had to obtain.
Once those functions were working locally, we implemented them on RosettaHub, from AWS.

3. Implementation on AWS

We used a web service - called Rosetta Hub - which uses a cluster of machines with multiple cores that unables us to run the program in a parallel fashion for the bigger target graphs.

We followed the next steps :

Step 1: The first step was to package the code in python script. We created a rdd.py and a df.py file for each version of the algorithm (files can be found in the appendix), as explained earlier. Note that in the df.py and rdd.py files, the output RDD with all the distinct nodes and their assigned groups is saved into a folder.

Step 2: We created an S3 storage bucket from the Rosetta Hub platform where we stored the two scripts and the files with graphs of different sizes. Indeed, from the target graph (web-Google.txt), we created a version containing only half of the lines, called web-Google_DEMI.txt and a version containing only a quarter of the lines, called web-Google_QUART.txt. We called the full / original version web-Google_FULL.txt. The objective was to compare the run time of the algorithms for files of different sizes.

Step 3: We launched the “Hadoop and Spark” Machine and opened the terminal.

Step 4: To upload the files from the S3 storage bucket to the local machine:

```
>>> aws s3 cp s3://s3-d56de376-fabd-4062-ac06-bb30a29ed634/file.csv file.csv
```

Then copied the files from local to HDFS :

```
>>> hdfs dfs -copyFromLocal file.csv file.csv
```

Step 5: To run our program :

```
>>> spark-submit df.py input_path  output_folder
```

Step 6: We followed the same steps backward to retrieve the files output by the programs.
Thanks to the access to the clusters, we were able to test our code on graphs of more important sizes.

III. Experimental analysis and scalability

One of the goals of this project is to compare the run-time performance of algorithms using two different Spark objects : RDD and DataFrames. To do so, we compare different sizes of graphs, for both algorithms. We decided to measure the size of the graph by the number of distinct edges.

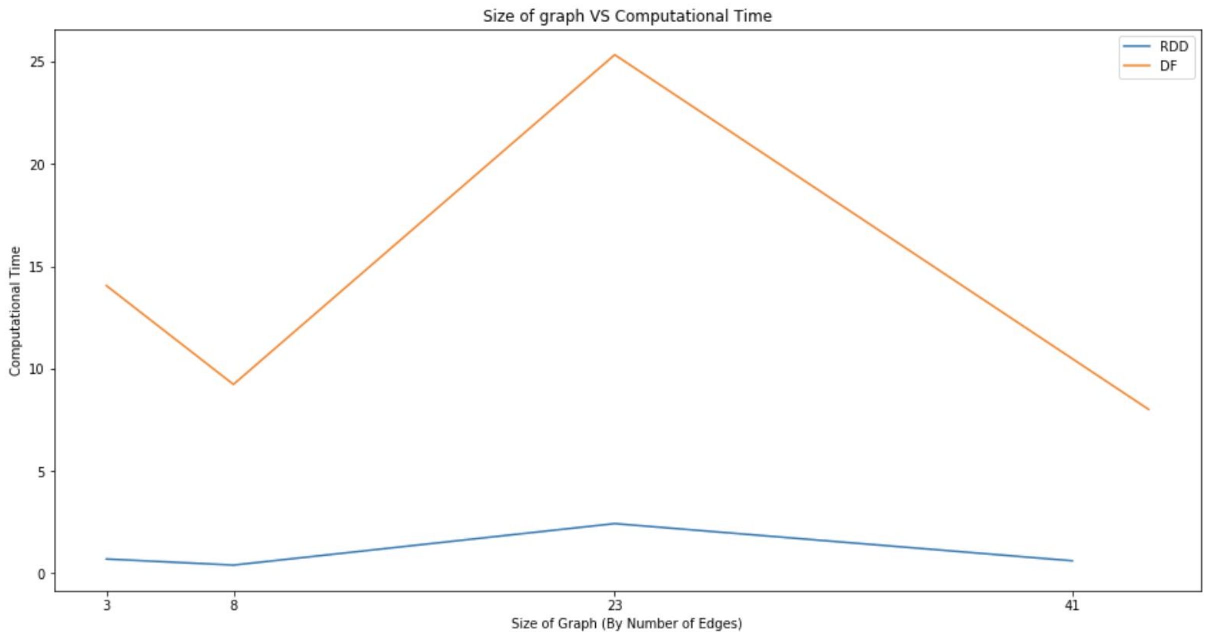
1. Experimental Phase

In the experimental phase of our project, we created 4 graphs of different sizes from scratch to be able to follow step by step our algorithms. As it was purely experimental, we designed the 4 graphs with pen and paper, wrote down the final results that we had to obtain, and then designed the RDDs / Dataframes representing those graphs to run the algorithms and test them.

Here is a summary of those graphs and the output of our algorithms :

	Graph_1	**Graph_2**	**Graph_3**	**Graph_4**
Num of Edges	3.000	8.000	23.000	41.000
Num of Groups	1.000	2.000	3.000	1.000
#of indirect neighboors	3.000	2.000	5.000	2.000
Computational time (RDD)	0.979	0.519	2.649	0.760
Computational time (DF)	13.684	7.404	24.715	10.718

To compare the local run-time performance of the algorithm using an RDD and the algorithm using a DataFrame, we plotted the following graph.



We can see that for the different sizes of graphs, the algorithm that uses dataframes always takes longer to run than the algorithm using RDDs.

We can also notice that the computational time is not directly related to the size of the graph (number of edges). This makes sense since the run time depends on the number of times the algorithm goes through our loop. During the equivalent of our Reduce function, we can group edges quickly, so even with a big number of edges, computation can go fast.

Our intuition is that the number of loops, and thus the computational time, depends on the level of indirecticity between nodes that are connected, which means that we are looking for the maximum distance between two connected points. As looping on the CCF_DEDUP_rdd() and

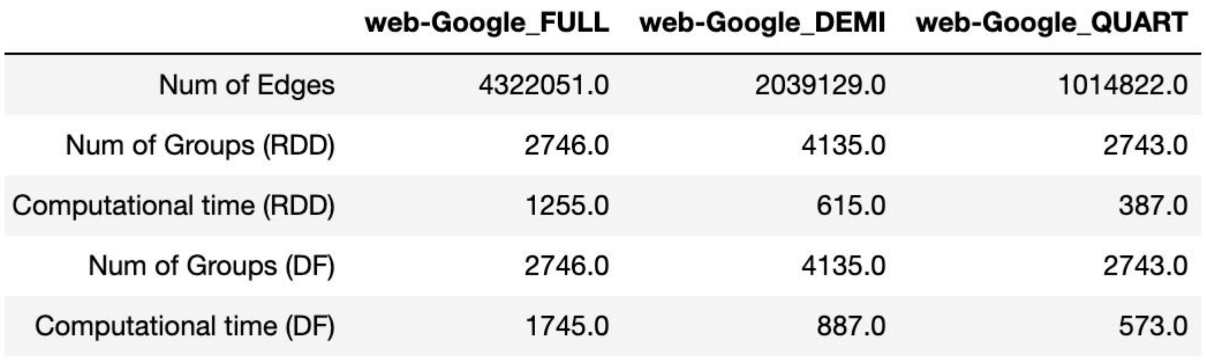
CCF_DEDUP_df() functions makes it possible to find nodes connected at step +1 compared to the previous loop, having a high level of indirecticity makes it necessary to loop more times.

However, the sizes of our graphs were small in the experimental phase, making it hard to conclude directly from them. We did so in the implementation phase.

2. Implementation Phase

For the implementation phase, we decided to use the same dataset used in the paper. It is a graph (web-google.txt) which was released in 2002 by Google as a part of Google Programming Contest. This dataset is a graph containing 875K nodes and 5.1M edges. Nodes represent web pages and direct edges represent hyperlinks between them.

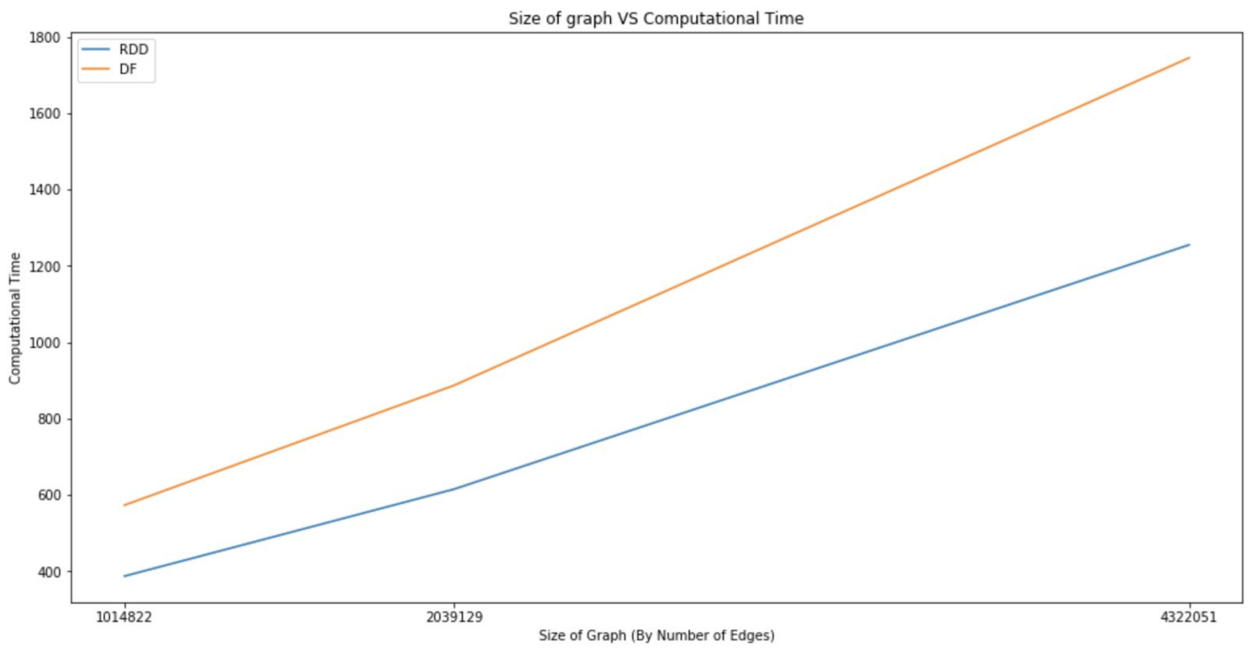
We followed the same logic as before for the implementation phase : We used the three graphs of different sizes from the initial graph that we described earlier (web-Google_FULL.txt, web-Google_DEMI.txt and web-Google_QUART.txt) to be able to compare the speed according to the size of the graph.
Here is a summary of the three graphs and their outputs :



	web-Google_FULL	web-Google_DEMI	web-Google_QUART
Num of Edges	4322051.0	2039129.0	1014822.0
Num of Groups (RDD)	2746.0	4135.0	2743.0
Computational time (RDD)	1255.0	615.0	387.0
Num of Groups (DF)	2746.0	4135.0	2743.0
Computational time (DF)	1745.0	887.0	573.0

The groups for each of the graphs can be found in the file sent with this report.

To compare the run-time performance on RosettaHub of the algorithm using an RDD and the algorithm using a DataFrame we plotted the following graph.



We obtain similar results as in the experimental phase. We clearly see that for the different sizes of graphs, the algorithm that uses dataframes always takes longer to run than the algorithm using RDDs.

Additionally, the run time looks linear and almost proportional to the size of the graph and is therefore not very scalable. We would have hoped for a logarithmic curve rather than a linear one.
However, we don't know the number of cores on our clusters that our machine was allowed to use, which may explain the lack of scalability.

IV. Strengths and weaknesses of our algorithms

Before elaborating on the strengths and weaknesses of our algorithms, we would like to point out how much we learnt by doing this project. Working on PySpark, with RDDs and Dataframes, was almost totally new to us, and we had the opportunity to understand how it worked, and the wide range of functionalities that we could use.
Also, we were able to run our algorithm on RosettaHub and upload the file on the platform - which was a big challenge in the beginning. We were very happy when we succeeded in doing so, as we looked back and saw all the steps we had taken to reach our objective.

1. Strengths

The main strength of our algorithms is that both versions (RDD and Dataframe) are easy to read. We focused on making our reasoning clear and explicit to the user. Granted, we have many steps within our functions, but the main objective was to be able to understand the code from scratch, without comments.

Another identified strength is the ability to use our local code on RosettaHub, which means that we can use it to run heavy files. This was the goal of re-writing functions that didn't exist on the platform.

2. Weaknesses

The main weakness of our algorithm is scalability. As said before, this should come from the size of our cluster in RosettaHub. If we had a bigger cluster, this problem would be solved.

As this was our first project using PySpark, rdd and dataframes, we are not sure we translated the MapReduce from the paper using the most effective techniques. However, the biggest graph we used which has 875K nodes and 5.1M edges ran in about 20 minutes using RDD and 30 minutes using DataFrames, which seems reasonable - even though we don't have a time reference to compare it with.

One axis of improvement on our code is the adaptability of our code. We could have added lines to catch errors. In this case, we knew exactly the format of the input and what it entailed. However, our code would crash if it was a bit different or if for instance there was a NaN.

V. Appendix - Copy of our notebook and .py files

Please find in our Google Drive the following folders:

- **1. Jupyter notebook and PDF version:** the full notebook and its PDF version
- **2. Python scripts for RosettaHub**: the rdd.py and df.py scripts that we used to implement our program on RosettaHub
- **3. Files - Web-Google**: the 3 input graphs of different sizes (txt files)
- **4. Output files - RDD**: for each of the 3 graphs of different sizes, the output RDDs - with the groups and their components - we obtained from running our program on RosettaHub
- **5. Output files - Dataframe:** for each of the 3 graphs of different sizes, the output Dataframe - with the groups and their components - we obtained from running our program on RosettaHub