CCF Project: Fast and Scalable Connected Component PySpark

November 2022

—

Slim Kachkachi, Arij Sakhaoui , Wided Labanne
Promotion IASD 4

Introduction

The hyperlink structure of the web (or web graph), social networks, and transportation networks (roads, bus routes, flights, ...) are only a few instances of how pervasive graphs are in contemporary culture. An elaborate metabolic and regulatory network, which can be visualized as a large, complex graph involving interactions between genes, proteins, and other cellular products, is essential to our very existence.

Graphs are made up of nodes and links (or edges) that connect pairs of nodes, with the possibility of having either directed or undirected connections. In some graphs we can find an edge from a node to itself, creating a self loop; in other cases, such edges are not allowed. Both nodes and graphs can have additional metadata. If we take the example of social networks: nodes representing individuals can also have attached demographic information like age or gender, and the edges can have type information (for example indicating type of relationship).

Finding connected components in a graph is crucial in many real world applications. From finding connections between social media members to advertising targeting. The work consists of understanding the MapReduce algorithm, and coding it into Spark by using both RDD and DataFrame *(coded in Python).*

In this project, we applied CCF (Connected Component Finder) to six different graphs of various sizes to test the scalability of the algorithm. We used databricks for small graphs, and the university cluster for the bigger ones. We also made tests using a Macbook Pro with an M1 chip.

1. A description of the adopted solution

What is MapReduce? Why graphs?

- The graph structure is represented with adjacency lists.
- Algorithms map over nodes and pass partial results to nodes on their adjacency lists. Partial results are aggregated for each node in the reducer.
- The graph structure itself is passed from the mapper to the reducer, such that the output is in the same form as the input.
- Algorithms are iterative and under the control of a non-MapReduce driver pro- gram, which checks for termination at the end of each iteration.

As mentioned in the introduction, MapReduce divides computing tasks into a map phase in which the input which is given as (key,value) pairs is split up among multiple machines to be worked on in parallel and a reduce phase in which the output of the map phase is put back together for each key to independently process the values for each key in parallel. Also, in a MapReduce context, recursion becomes iteration.

What we want to achieve here is to find all connected components in a graph.

For simplicity reasons, we will use the smallest node id in each connected component in order to identify that component.

The algorithm we're implementing, Connected Component Finder, takes as an input the list of all the edges in the graph, and returns the mapping from each node in the graph to its corresponding component ID, that is the smallest node-id in its corresponding connected component.

2. Designed algorithms

To reach the desired outcome, we have implemented two MapReduce jobs: *CCF-Iterate* and *CCF-Dedup*. These jobs will run iteratively until we have all corresponding component Ids for all the nodes in the graph.

***CCF-Iterate*: link nodes to component ID**

Description: The CCF-iterate job generates adjacency lists AL = (a_1, a_2, ..., a_n) for each node v, and if the node id of this node v_id is less than the minimum node id a_min in the adjacency list, we will not output a pair. Otherwise, it first creates a pair (v_id , a_min) and then a pair for each (a_i,a_min), where a_i in AL and a_i <> a_min. If there is only one node in AL, it means we generate the pair we have in the previous iteration. However, if there is more than one node in AL, it means that we might generate a pair that we didn't have in the previous iteration, and another iteration is needed.

Input:

-For the first iteration: initial edge list

- For the rest: the output of *CCF-Dedup* from previous iterations

CCF-Dedup: dump copies of outputs of *CFF-Iterate*

Description: CCF-Dedup job duplicates the output of CCF-iterate job. We use this job because it increases the efficiency of CCF-iterate job in terms of speed and overhead.

Input:

-Output of of the CCF-iterate job

***CCF-Iterate with secondary sorting**:* link nodes to component ID.

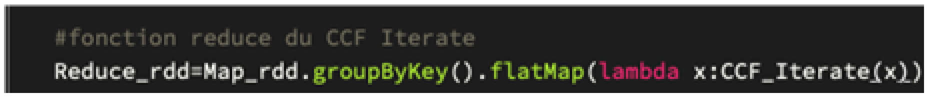
Description: This version of CCF-Iterate job is meant to improve the space complexity. Instead of iterating through the entire list, values can be passed to the reducer in a sorted way with custom partitioning. The difference this algorithm makes is noticeable in real world graphs with billions of nodes.

Input: same as CCF-Iterate job.

3. Comments to main fragments of code

The main code function *CCF_Iterate* constitutes the adjacency list of the graph nodes. The counter management of pairs number, created at each iteration, also required a different choice of implementation between RDD and DF versions.

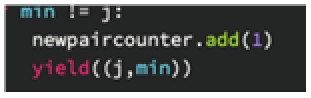
For the RDD code, we chose to use the combination of the operators "*groupByKey.flatMap(lambda x : CCF_Iterate(x))*" instead of the *reduceByKey()* (the operation performed by Python function not being associative) to trigger the execution of the *CFF_Iterate function* (see illustration n°1).



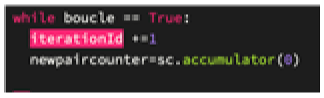
```
#fonction reduce du CCF Iterate
Reduce_rdd=Map_rdd.groupByKey().flatMap(lambda x:CCF_Iterate(x))
```

illustration n°1

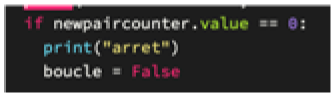
Regarding the counter set up, we created an accumulator with the "*accumulator*" operator which allows to manage a global variable (initialization, increment) transverse to the various tasks on the "*executors*" (cf. illustration n°2).



```
min != j:
  newpaircounter.add(1)
  yield((j,min))
```



```
while boucle == True:
  iterationId +=1
  newpaircounter=sc.accumulator(0)
```

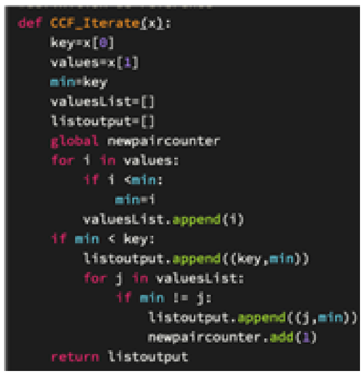


```
if newpaircounter.value == 0:
  print("arret")
  boucle = False
```

Illustration n°2

Nevertheless its implementation was tested according to two approaches:

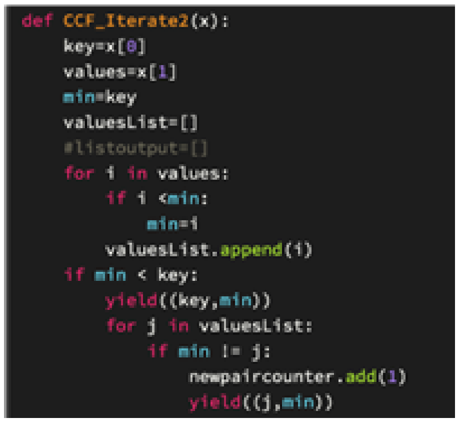
- The first approach returned a python list by the "*return*" operator (illustration n° 3);



```
def CCF_Iterate(x):
    key=x[0]
    values=x[1]
    min=key
    valuesList=[]
    listoutput=[]
    global newpaircounter
    for i in values:
        if i <min:
            min=i
        valuesList.append(i)
    if min < key:
        listoutput.append((key,min))
        for j in valuesList:
            if min != j:
                listoutput.append((j,min))
                newpaircounter.add(1)
    return listoutput
```

Illustration n°4

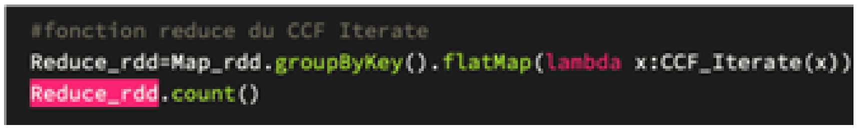
- The 2nd approach used the "yield" operator which returns the pairs created in the course of the process (illustration n°4).



```python
def CCF_Iterate2(x):
    key=x[0]
    values=x[1]
    min=key
    valuesList=[]
    #listoutput=[]
    for i in values:
        if i <min:
            min=i
        valuesList.append(i)
    if min < key:
        yield((key,min))
        for j in valuesList:
            if min != j:
                newpaircounter.add(1)
                yield((j,min))
```

Illustration n°4

We noticed that there is no major difference in terms of execution time, as long as we don't use count() as it slows down dramatically the execution time (illustration n° 5 shows the first version of our code).

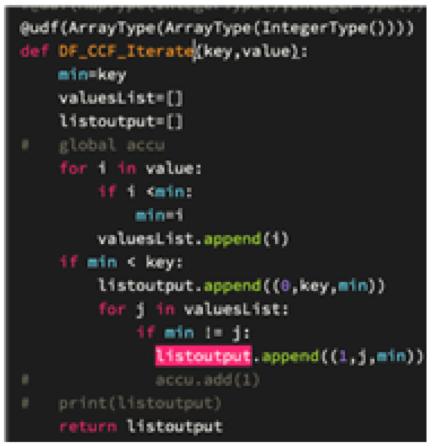


```python
#fonction reduce du CCF Iterate
Reduce_rdd=Map_rdd.groupByKey().flatMap(lambda x:CCF_Iterate(x))
Reduce_rdd.count()
```

Illustration n°5

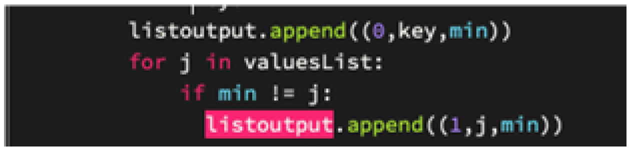
Speed comparisons between these two codes are presented in the experimental part.

For DF code, we chose to use a "*User Defined Function*" (UDF) to trigger the execution of the *CFF_Iterate* (see figure 6). The management of the counter of created pairs was carried out by returning a triplet whose 1st value is '0' or '1' corresponding to the creation or not of a pair at each iteration of this function (illustration n° x-7). It is sufficient to apply a filter on this first column and to count the number of lines of this new DataFrame to obtain the number of created pairs (cf. illustration n°8).

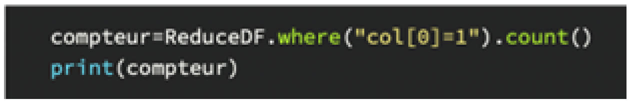


```
@udf(ArrayType(ArrayType(IntegerType())))
def DF_CCF_Iterate(key,value):
    min=key
    valuesList=[]
    listoutput=[]
#   global accu
    for i in value:
        if i <min:
            min=i
        valuesList.append(i)
    if min < key:
        listoutput.append((0,key,min))
        for j in valuesList:
            if min != j:
              listoutput.append((1,j,min))
#             accu.add(1)
#   print(listoutput)
    return listoutput
```

Illustration n°6



```
listoutput.append((0,key,min))
for j in valuesList:
    if min != j:
      listoutput.append((1,j,min))
```



```
compteur=ReduceDF.where("col[0]=1").count()
print(compteur)
```

Illustration n°7

Illustration n°8

4. Experimental analysis

The scalability assessment of the codes and platforms consisted in a comparison of time execution according to 4 criteria:

- The dataset size and number of edges (0.8MB for 800K edges to 500MB for 34M edges) (1);
- Size / power of the Spark environments used (MAC Book, Google Dataproc, Lamsade cluster, DataBricks and Google Collab) (2);
- RDD VS DF versions;
- Spark environment set up to maximize parallelism ("partitioning", "executor").

Note that the platforms' performances vary greatly from one time to another (Providing the same availability of resources could be one explanation). We will therefore focus on orders of magnitude.

Within Google Cloud and Lamsade environments we used the command "*spark-submit filename.py*" to execute the codes. In the particular case of the Lamsade cluster, we also used the options "*spark-submit -executor-cores x -num-executors y filename.py*" to adjust on the number of cores and executors.

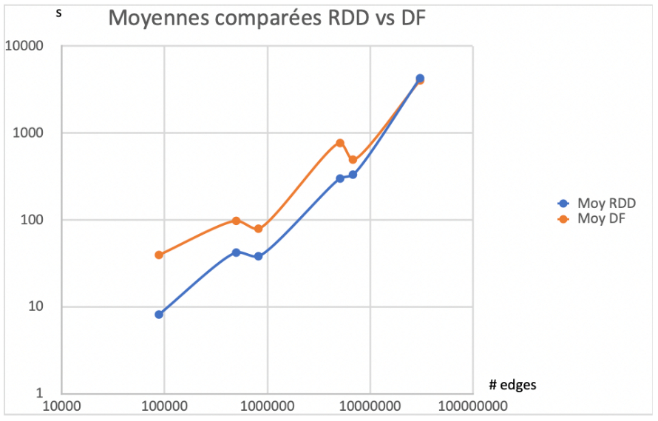
With regards to RDD code versions, we worked on two versions of the code with an optimisation goal and compared related performances.

Finally, we tried to implement CCF_Iterate with Secondary Sorting without success. We mentioned in the appendix the difficulties we faced implementing that algorithm.

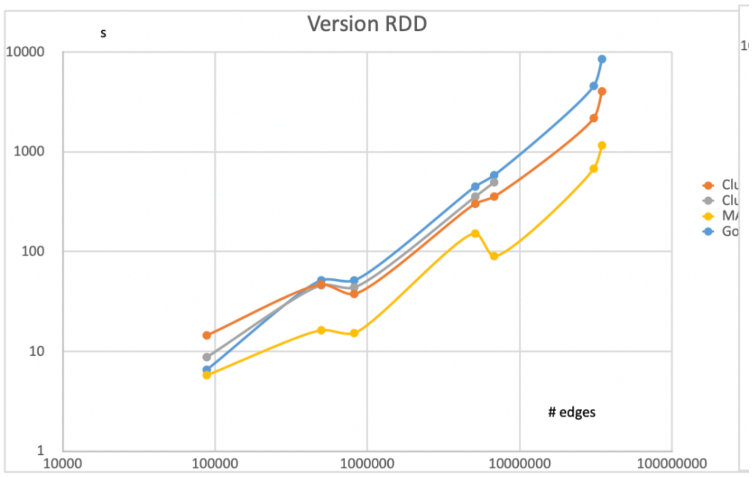
We focused on the performances of both RDD and DF codes with a difference of the number of edges(1) and environments(2), leaving to each default parameters (number of partitions, number of cores etc.).

<u>The first observation is that the RDD code was faster than the DF code</u> (graph 1). On logarithmic scales, execution times are relatively linear for the RDD code with respect to the number of edges (graph 2) whereas they seem to grow faster than the file size for the DF version (graph 3).

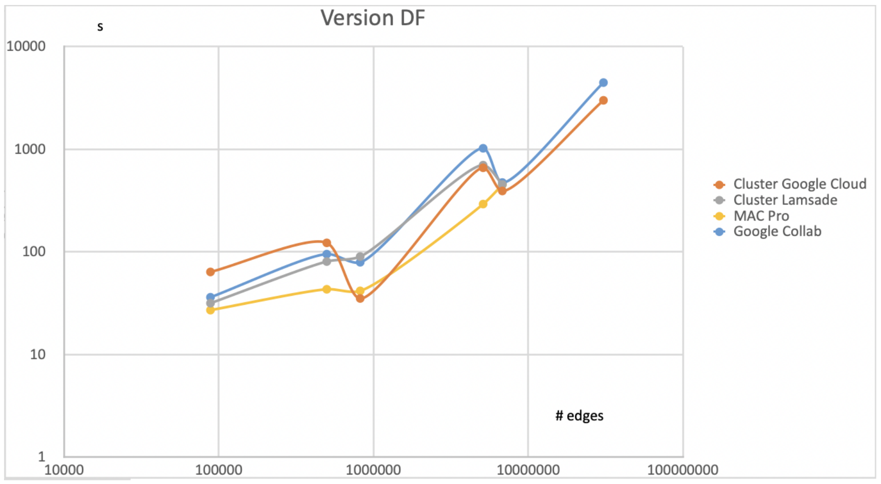
The graphs show two thresholds at which the performances improve punctually on the different platforms. It was not possible to interpret these phenomena. A possible explanation could be the difference in "Connected Components", since these two levels correspond to similar sets in size (octets) but not in number of "Connected Components".



Graphe n° 1



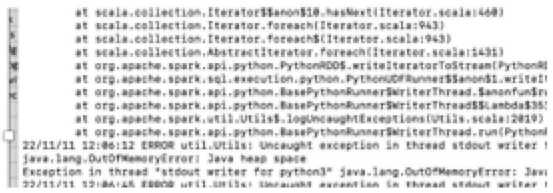
Graphe n° 2



Graphe n° 3

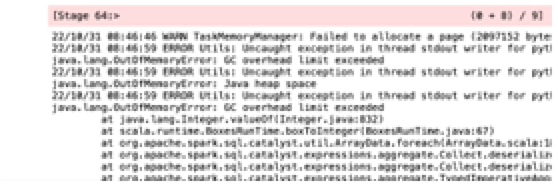
<u>As a second observation, we noticed that execution times on the Spark environment of the MacPro were shortest for all datasets</u>. Question mark on this observation, but we figure out that datasets (even the 500MB one) are finally relatively small to fully benefit from Spark's parallelism (without the initial startup phase with data exchanges on the network which is relatively slow).

<u>A 3rd observation tells that for biggest datasets (Soc_Pokec - 424 MO and 30M edges), the codes applied to DF resulted to "*out of memory*" messages</u> on the MacPro and on the Lamsade Cluster (see illustrations n° x & n° x), unlike RDD versions.



```
        at scala.collection.Iterator$$anon$10.hasNext(Iterator.scala:460)
        at scala.collection.Iterator.foreach(Iterator.scala:943)
        at scala.collection.Iterator.foreach$(Iterator.scala:943)
        at scala.collection.AbstractIterator.foreach(Iterator.scala:1431)
        at org.apache.spark.api.python.PythonRDD$.writeIteratorToStream(PythonR
        at org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.writeI
        at org.apache.spark.api.python.BasePythonRunner$WriterThread.$anonfun$r
        at org.apache.spark.api.python.BasePythonRunner$WriterThread$$Lambda$35
        at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:2019)
        at org.apache.spark.api.python.BasePythonRunner$WriterThread.run(Python
22/11/11 12:06:12 ERROR util.Utils: Uncaught exception in thread stdout writer 
java.lang.OutOfMemoryError: Java heap space
Exception in thread "stdout writer for python3" java.lang.OutOfMemoryError: Jav
22/11/11 12:06:45 ERROR util.Utils: Uncaught exception in thread stdout writer 
```

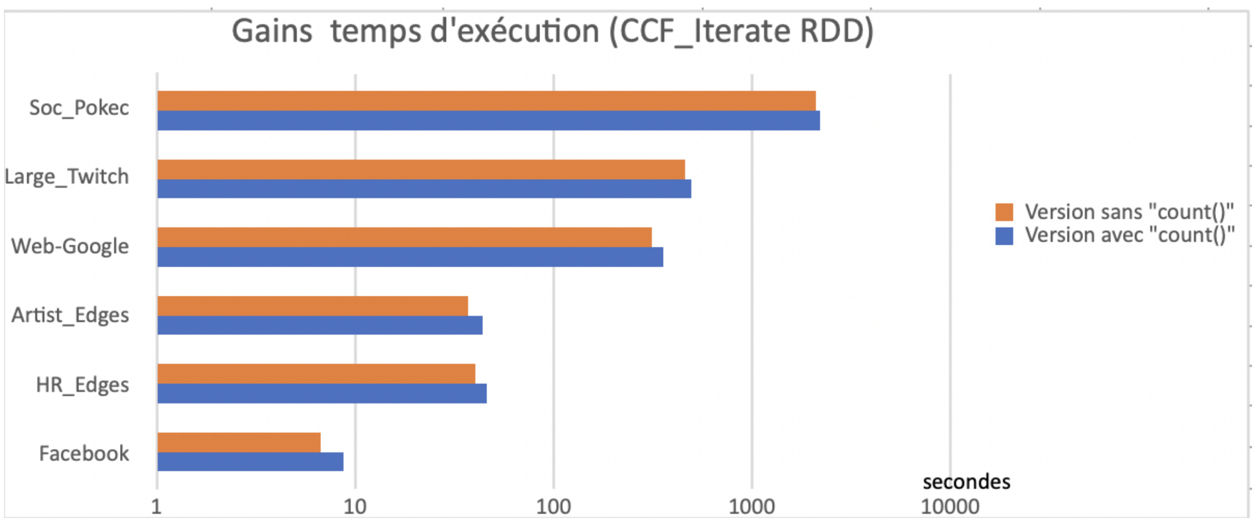
Illustration n°9



```
[Stage 64:>                                                          (0 + 8) / 9]
22/10/31 08:46:46 WARN TaskMemoryManager: Failed to allocate a page (2097152 byte
22/10/31 08:46:59 ERROR Utils: Uncaught exception in thread stdout writer for pyt
java.lang.OutOfMemoryError: GC overhead limit exceeded
22/10/31 08:46:59 ERROR Utils: Uncaught exception in thread stdout writer for pyt
java.lang.OutOfMemoryError: Java heap space
22/10/31 08:46:59 ERROR Utils: Uncaught exception in thread stdout writer for pyt
java.lang.OutOfMemoryError: GC overhead limit exceeded
        at java.lang.Integer.valueOf(Integer.java:832)
        at scala.runtime.BoxesRunTime.boxToInteger(BoxesRunTime.java:67)
        at org.apache.spark.sql.catalyst.util.ArrayData.foreach(ArrayData.scala:1
        at org.apache.spark.sql.catalyst.expressions.aggregate.Collect.deserializ
        at org.apache.spark.sql.catalyst.expressions.aggregate.Collect.deserializ
        at org.apache.spark.sql.catalyst.expressions.aggregate.TypedImperativeAgg
```

Illustration n°10

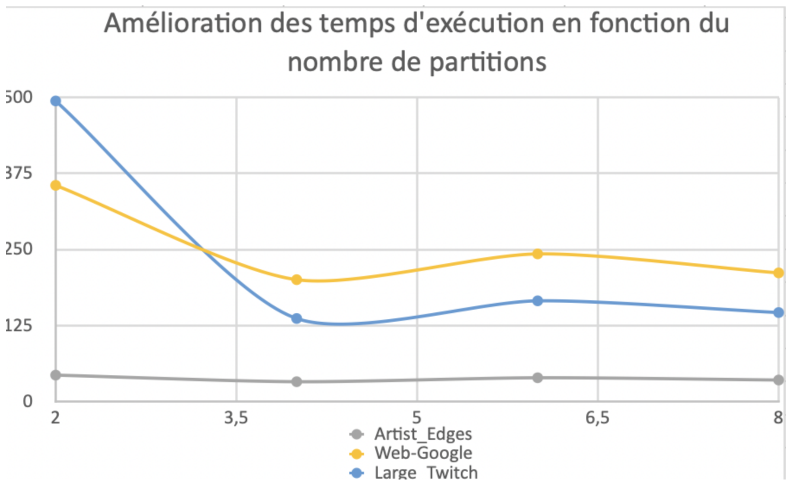
<u>Lastly, we compared the relative performances of the two versions of code on RDD with regard to the CCF_Iterate function</u>. The removal of the "count()" action resulted to a marginal improvement in time execution (see graph n/ .) ;



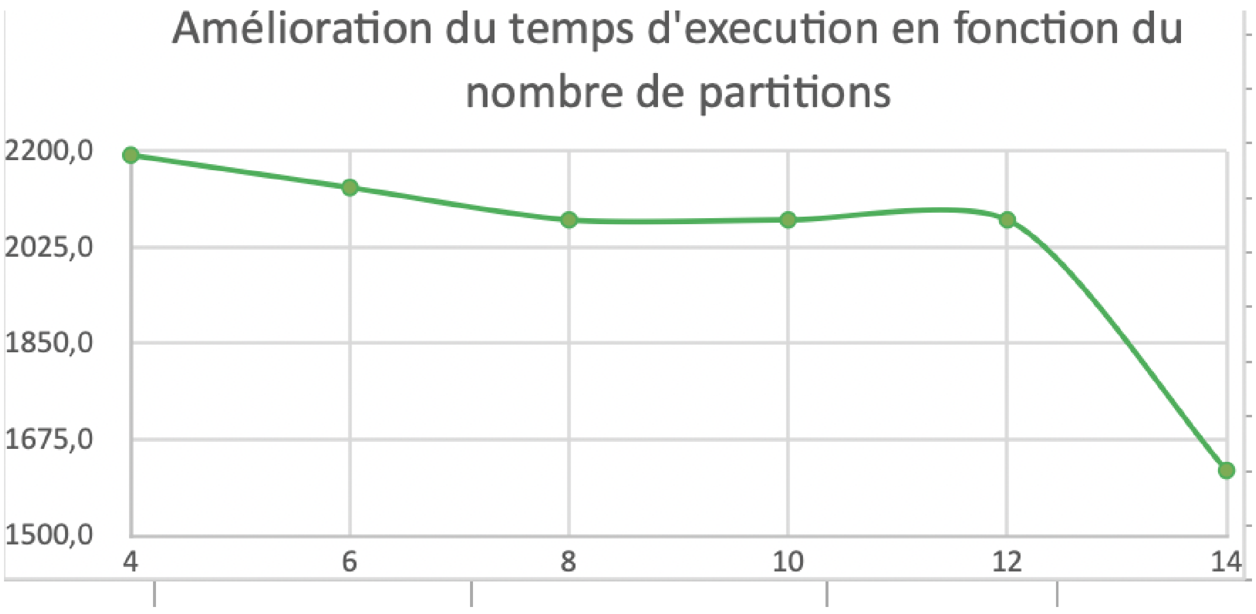
Graphe n°4

We then studied the **impact of both the partitioning and the spark environment settings on the execution time.** The first trial tested several increasing values of partitioning from the default values proposed by the Lamsade cluster (1).

The graph on the left below shows the reduction in execution time for small datasets (up to 70 MB) and the one on the right for the 424 MB dataset. For small datasets the effect is significant for a passage from 2 to 4 partitions; beyond that the effect is null or even counter-productive. For the 424 MB dataset, it was necessary to go up to 14 partitions to observe a positive effect.

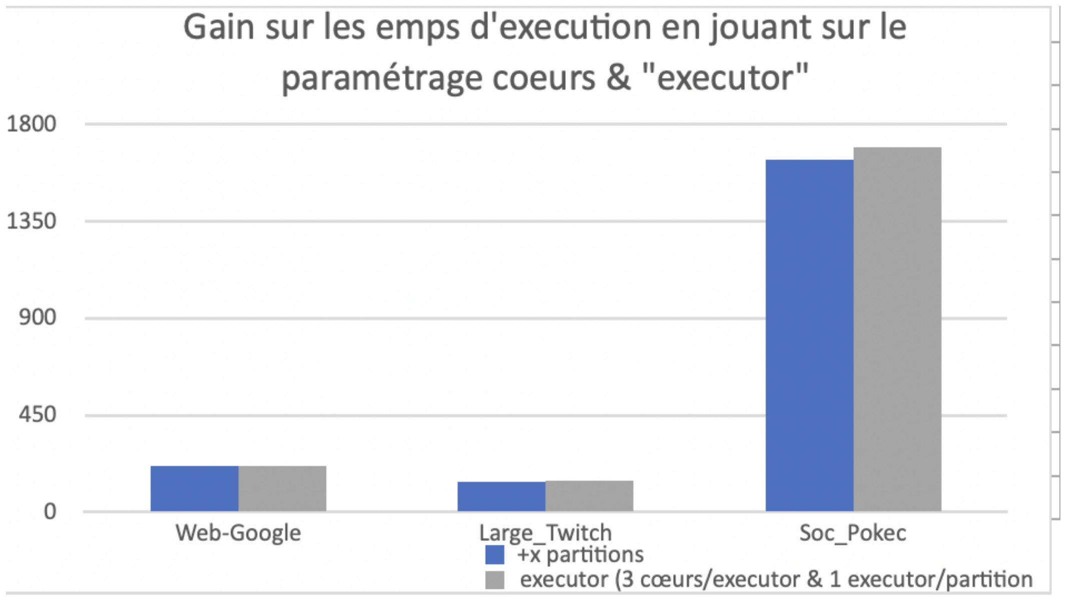


Graphe n° 6



Graphe n° 5

Finally, a last test was carried out by playing with the number of cores and executors, starting from the averages recommended in the spark documentation (3 cores per executor). The results on several data sets with 30M edges (and 424 MB) show no gain (see graph n° x). Our analysis is that the main factor is the choice of the number of partitions according to the size of the dataset and that we did not find the right setting for the other parameters.



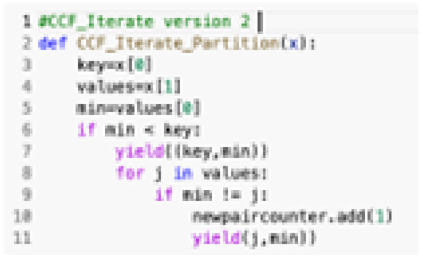
Graphe n° 7

Appendix : difficulty encountered in the implementation of the "secondary sort" option of the "CCF_Iterate

We have tried to implement the version of CCF_Iterate that does not require the value to be traversed to identify the minimum of the adjacent list.
The principle of this approach is to submit to the CCF_Iterate a list of nodes sorted in ascending order.
To achieve this goal two actions are necessary:

1. To group all the "values" of the same "key" in the same partition (see illustration n°)
2. To apply a partitioning method that natively sorts all the values of a key.

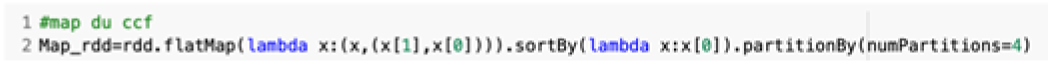
The code of CFF_Iterate becomes very reduced (see illustration n°x)



```
#CCF_Iterate version 2
def CCF_Iterate_Partition(x):
    key=x[0]
    values=x[1]
    min=values[0]
    if min < key:
        yield((key,min))
        for j in values:
            if min != j:
                newpaircounter.add(1)
                yield(j,min))
```

Illustration n°11

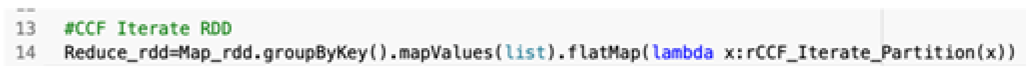
The first step consists in assigning the pairs by partition (we have chosen here 4).



```
#map du ccf
Map_rdd=rdd.flatMap(lambda x:(x,(x[1],x[0]))).sortBy(lambda x:x[0]).partitionBy(numPartitions=4)
```

Illustration n° 12

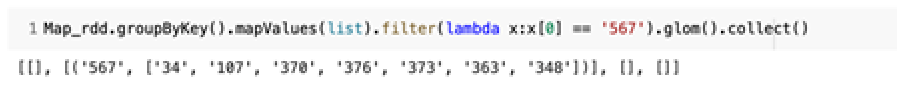
Then to pass the adjacent lists to CFF_Iterate as shown in the following figure.



```
#CCF Iterate RDD
Reduce_rdd=Map_rdd.groupByKey().mapValues(list).flatMap(lambda x:rCCF_Iterate_Partition(x))
```

Illustration n° 13

But this approach does not work. If the Key and values are well grouped by partition, we could verify that spark does not respect the order of the values in the adjacent list as shown in the following figure. The documentation reveals that it is actually necessary to create a specific partitioning class using the partitionBy() command.



```
Map_rdd.groupByKey().mapValues(list).filter(lambda x:x[0] == '567').glom().collect()
```

```
[[], [('567', ['34', '107', '370', '376', '373', '363', '348'])], [], []]
```

Illustration n° 14

dataset	Edges number	Size (MO)	CC number	Partitioning (default)
Facebook_c	88234	0,85	1	2
HR_Edges	498202	6,2	6	2
Artist_Edges	819306	10,2	3	2
Web-Google	5107039	75,4	2746	2
Large_Twitch	6797557	86,2	1	2
Soc_Pokec	30622564	424	1	4
com-LiveJournal	34681189	500	1	4

Appendix description of the data sets used

Google Collab	No info
Databricks	2 work nodes without precision on the dimensions
Google Cloud	1 master and 2 work nodes. 500 GO disk each Master n1-std-1 not specified Work node (2 virtual processors 7,5 GB Ram) Block size 134217728
Cluster Lamsade	2 work nodes ? / possibility to play on the number of cores and executors
Mac Book pro	M1 chip (8 cores) 16 GB RAM Partitioning from 2 to 14

Appendix: descriptions of the spark configuration

DF		taille en MO		Nbre itération	Nbre de CC	Google Collab	DataBrick	Cluster Google Cloud	Cluster Lamsade
Facebook	1		88234	5	1	36,2	38,2	63,9	31,7
HR_Edges	2		498202	6	6	95,3	144,9	122,7	80,7
Artist_Edges	3		819306	6	3	79,4	148,8	35,25	90,3
Web-Google	4		5107039	8	2746	1023,7	1150	661,2	701,4
Large_Twitch	5		6797557	6	1	475,23	694,2	393	454,5
Soc_Pokec	6		30622564	6	1	4470	4590	3002,5	Out Of Memory

Experimental analysis comparing the RDD and DataFrame versions has to be conducted on graphs of increasing size

For small graphs use Databricks, for bigger ones use the cluster

5. Strong points and limitations

Strong points:

Connected Component Finder is an efficient algorithm that helps find all the connected components in a graph. It is a fairly easy algorithm to both understand and implement. It is implemented in the MapReduce framework and does not use a lot of memory which makes it scalable to graphs with billions of nodes and edges.

Limitations:

CCF uses MapReduce Framework, and as discussed earlier, that comes with upsides and downsides.

The key distinction between single-machine graph algorithms and MapReduce graph algorithms is that the latter can generally be stored in memory for quick, random access. Clearly, this is not feasible with MapReduce as there is no inbuilt method for exchanging global state in the programming model. Communication can only take place from a node to the nodes it links to, or from a node to a node from nodes linked to it, as the most natural representation of large sparse graphs is with adjacency lists; in other words, information passing is only feasible within the local graph structure.

This limitation gives rise to the structure of many graph algorithms in MapReduce: local computation is carried out on each node, and the outcomes are "passed" to its neighbors. Convergence on the global graph is possible after several iterations. The MapReduce execution framework's shuffling and sorting capabilities enable the sending of partial results along a graph edge. In this algorithm, we assumed that all graphs are sparse graphs since the quantity of intermediate data generated is on the order of the number of edges. Copying intermediate data across the network would account for a large part of the MapReduce running time for dense graphs, which in the worst scenario would equal the graph's node count O(n2).

So MapReduce algorithms are often impractical on dense graphs.

Summary:
In this project, we applied two versions (using DataFrames and RDD) of CCF (Connected Component Finder) to six different graphs of various sizes to test the scalability of the algorithm, and found some interesting results after changing both the partitioning and the spark environment. The algorithm was efficient and easy to implement. However, we have not tested the secondary sorting approach due to technical difficulties.

What’s next?

CCF was the first approach that employed the concept of node exclusions, although limited to the seed nodes However, the impact is negligible as the number of seed nodes is usually much lower than the nodes of a graph.

After doing some research, we have found an interesting paper explaining CRACKER, that employs node exclusion in an extensive manner by processing only the relevant vertices, while discarding vertices that have no useful information to share.

The strategy of CRACKER is to transform the input graph into a set of trees, one for each connected component in the graph. Nodes are iteratively removed from the graph and added to the trees, reducing the amount of computation at each iteration.

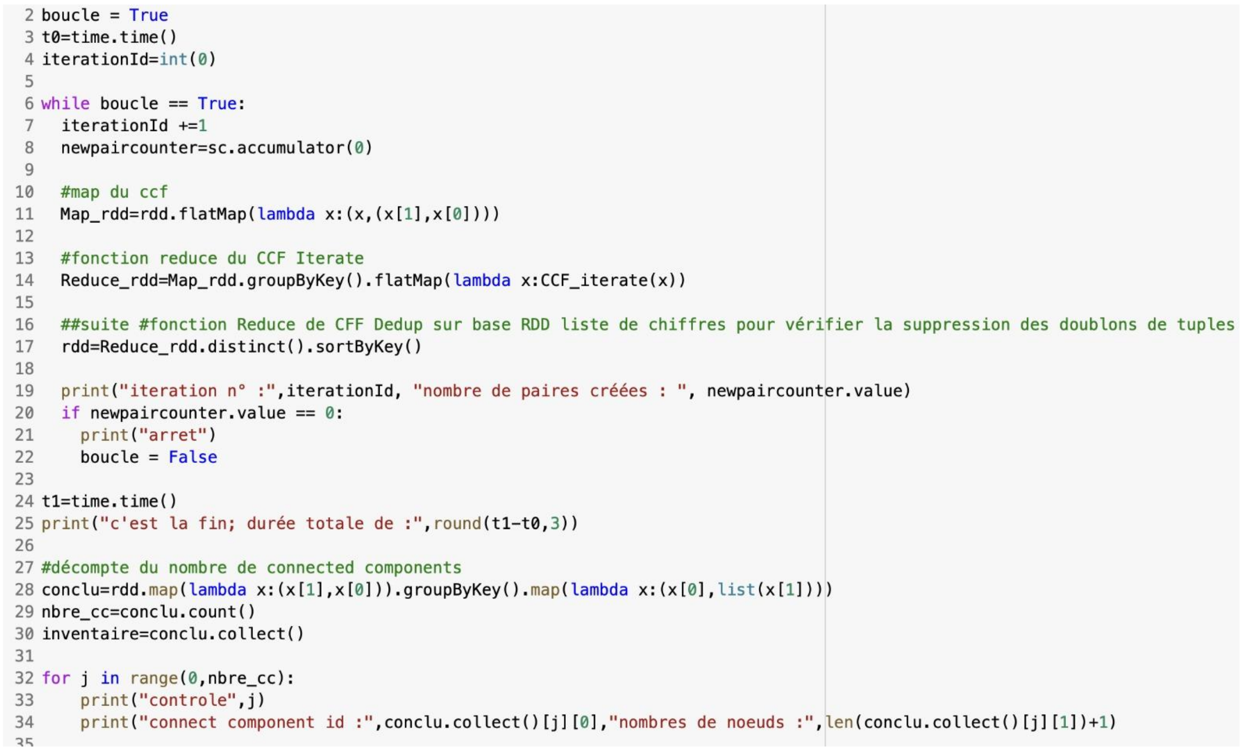
The CRACKER algorithm achieves the identification of the connected components in two phases:

• Seeds Identification: CRACKER identifies the seed nodes of the graph, and iteratively builds a spanning tree for each CC, rooted in its seed; whenever a node is added to the spanning tree, it is excluded from computation in the subsequent iterations

• Seeds Propagation: propagates the seed id to all the nodes belonging to the CC by exploiting the spanning tree built in the previous phase.

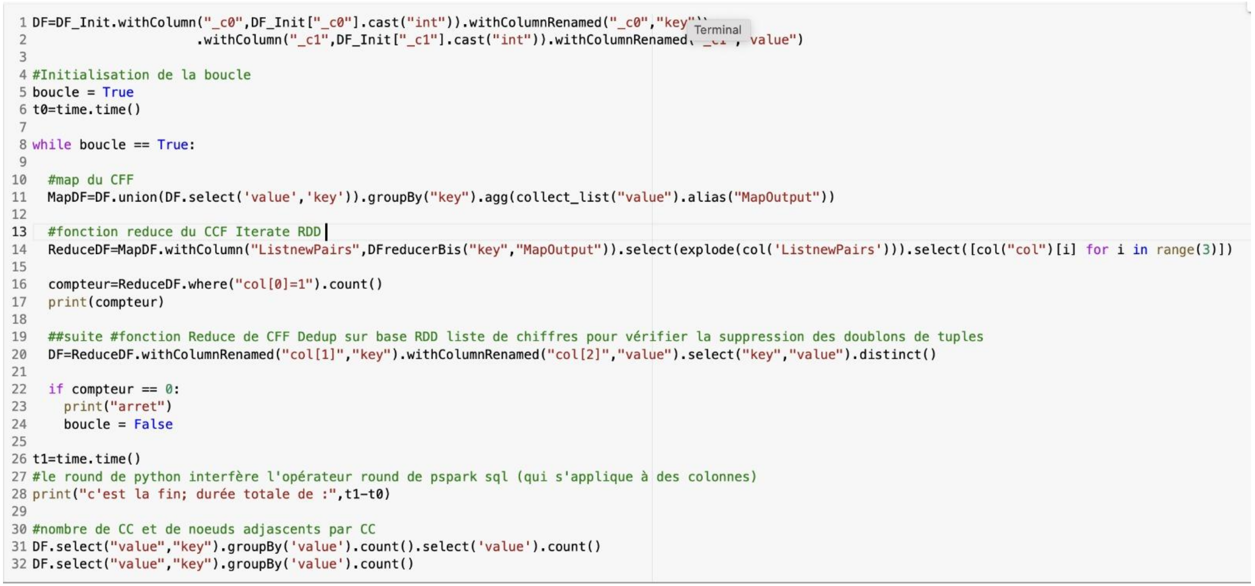
In the future, we will implement this algorithm and compare our CCF results with those of CRACKER.

6. Appendix



```
boucle = True
t0=time.time()
iterationId=int(0)

while boucle == True:
  iterationId +=1
  newpaircounter=sc.accumulator(0)

  #map du ccf
  Map_rdd=rdd.flatMap(lambda x:(x,(x[1],x[0])))

  #fonction reduce du CCF Iterate
  Reduce_rdd=Map_rdd.groupByKey().flatMap(lambda x:CCF_iterate(x))

  ##suite #fonction Reduce de CFF Dedup sur base RDD liste de chiffres pour vérifier la suppression des doublons de tuples
  rdd=Reduce_rdd.distinct().sortByKey()

  print("iteration n° :",iterationId, "nombre de paires créées : ", newpaircounter.value)
  if newpaircounter.value == 0:
    print("arret")
    boucle = False

t1=time.time()
print("c'est la fin; durée totale de :",round(t1-t0,3))

#décompte du nombre de connected components
conclu=rdd.map(lambda x:(x[1],x[0])).groupByKey().map(lambda x:(x[0],list(x[1])))
nbre_cc=conclu.count()
inventaire=conclu.collect()

for j in range(0,nbre_cc):
    print("controle",j)
    print("connect component id :",conclu.collect()[j][0],"nombres de noeuds :",len(conclu.collect()[j][1])+1)
```

Appendix : RDD code body (the code of the CCF_Iterate function is available in the analysis of this report)



```
DF=DF_Init.withColumn("_c0",DF_Init["_c0"].cast("int")).withColumnRenamed("_c0","key"))
                     .withColumn("_c1",DF_Init["_c1"].cast("int")).withColumnRenamed("_c1","value")

#Initialisation de la boucle
boucle = True
t0=time.time()

while boucle == True:

  #map du CFF
  MapDF=DF.union(DF.select('value','key')).groupBy("key").agg(collect_list("value").alias("MapOutput"))

  #fonction reduce du CCF Iterate RDD
  ReduceDF=MapDF.withColumn("ListnewPairs",DFreducerBis("key","MapOutput")).select(explode(col('ListnewPairs'))).select([col("col")[i] for i in range(3)])

  compteur=ReduceDF.where("col[0]=1").count()
  print(compteur)

  ##suite #fonction Reduce de CFF Dedup sur base RDD liste de chiffres pour vérifier la suppression des doublons de tuples
  DF=ReduceDF.withColumnRenamed("col[1]","key").withColumnRenamed("col[2]","value").select("key","value").distinct()

  if compteur == 0:
    print("arret")
    boucle = False

t1=time.time()
#le round de python interfère l'opérateur round de pspark sql (qui s'applique à des colonnes)
print("c'est la fin; durée totale de :",t1-t0)

#nombre de CC et de noeuds adjascents par CC
DF.select("value","key").groupBy('value').count().select('value').count()
DF.select("value","key").groupBy('value').count()
```

Annexe : DF code body (the code of the CCF_Iterate function is available in the analysis of this report)