

Rapport BIG DATA

Master 2 MIAAGE ID App

Fast and Scalable Connected Component Computation in MapReduce

Présenté par :

Riad FEZZOUA

Etienne EDY

Karim Moumene AMROUCHE

Encadré par :

Dario COLAZZO

Sommaire

Introduction	1
1 Algorithme initial	2
1.1 Présentation	2
1.2 Aperçu de l'algorithme	3
1.3 Analyse des performances	4
2 Algorithme Transitoire	5
2.1 Présentation	5
2.2 Aperçu de l'algorithme	6
2.3 Analyse des performances	6
3 Algorithme Final	7
3.1 Présentation	7
3.2 Aperçu de l'algorithme	8
3.3 Analyse des performances	9
3.3.1 Sur Databricks	10
3.3.2 Sur RosettaHub	12
Conclusion	14

Introduction

L'émergence du numérique rend le monde qui nous entoure de plus en plus connecté. De cette connexion omniprésente sont nées les données numériques. Le principe du big data réside dans la collecte et le stockage massif de ces données dans une optique d'analyse. Cela permet une analyse plus fine du monde qui nous entoure.

L'enjeu est immense et les opportunités très variées, l'exploitation de ces données est devenu un atout majeur pour ceux qui la maîtrisent. On pourrait citer comme exemple l'amélioration des performances et processus de production ou même l'anticipation et gestion des crises et risques.

La modélisation et la fouille des données est aujourd'hui un axe de recherche important. Il existe plusieurs types de modélisation et parmi eux : les bases de données orientées graphes. Ce modèle est très utilisé, notamment dans la détection de fraude et dans les systèmes de recommandation.

Les bases de données orientées graphe ont recours à des nœuds pour stocker les entités de données, ainsi qu'à des périphéries pour stocker les relations entre les entités. Une périphérie possède toujours un nœud de départ, un nœud de fin, un type et une direction. Un nœud peut décrire les relations, actions, possession parent-enfant, etc. Le nombre et le type de relations qu'un nœud peut avoir sont illimités.

Dans le cadre de ce projet, nous mettons en pratique un algorithme de recherche de composantes connexes dans un graphe sur Spark. Nous présenterons différentes implémentations avec leurs performances et analyserons les résultats obtenus. Les algorithmes ont été testés sur les plateformes Data bricks et RosettaHub, avec des graphes modélisant des réseaux sociaux tels que Facebook ou Twitter.

Algorithme initial

1.1 Présentation

Nous avons abordé ce sujet de manière simpliste dans un premier temps. C'est à dire que nous avons d'abord cherché à développer un algorithme résolvant le problème des composantes connexes de la manière la plus simple possible.

C'est cet algorithme que nous allons vous présenter à présent. Nous avons choisi de diviser l'algorithme de CCF en trois fonctions :

- la première, appelée *duplicate* permettant de créer tous les couples nécessaires lors de chaque itération du CCF.
- la seconde, appelée *ccf iterate* qui reproduit une itération de l'algorithme CCF
- la troisième appelée *résolution algorithme* qui permet d'effectuer le bon nombre d'itération en fonction de la valeur de *New Pair Count* (cpt dans notre code) renvoyée à chaque itération.

1.2 Aperçu de l'algorithme

```
1 def duplicate(rdd):
2     return rdd.union(rdd.map(lambda x : (x[1],x[0])))
3
4 def ccf_iterate(rdd):
5     liste = []
6     cpt = 0
7     value_list = []
8     r = duplicate(rdd).groupByKey().mapValues(list)
9     for i in range(r.count()):
10        value_list = []
11        key = r.collect()[i][0]
12        min_v = key
13        for value in r.collect()[i][1]:
14            if (value < min_v):
15                min_v = value
16            value_list.append(value)
17        if(min_v < key):
18            x = (key, min_v)
19            if x not in liste:
20                liste.append(x)
21        for j in value_list:
22            if(min_v != j):
23                cpt = cpt+1
24                y = (j,min_v)
25                if y not in liste:
26                    liste.append(y)
27    rdd_answer = spark.sparkContext.parallelize(liste)
28    return rdd_answer,cpt
```

```
1 def resolution_algorithme(rdd):
2     cpt=1
3     while(cpt!=0):
4         rdd,cpt=ccf_iterate(rdd)
5     return rdd
```

1.3 Analyse des performances

En termes de résultat, il va sans dire que nous n'attendions pas cet algorithme au rendez-vous. Et en effet, ses performances n'ont pas été extraordinaires. Pour vous donner un exemple, il réalisait l'exemple donnée dans l'énoncé en 35 secondes.

Néanmoins, cet algorithme nous a permis de comparer les résultats de nos algorithmes futurs et a été d'une grande aide lors de la création des prochains algorithmes.

De plus, la longueur de son code nous a permis de bien comprendre chacune de ces étapes et donc d'appréhender au mieux le problème des composantes connexes.

Algorithme Transitoire

2.1 Présentation

Voici présentée ci-dessous une deuxième version de l'algorithme de base. L'objectif est d'améliorer l'efficacité de la première version.

Le concept de cet algorithme est simple : au début, il trie la liste "valeurs" afin d'avoir pour chaque clé, un accès direct à la valeur minimum parmi toutes celles pouvant potentiellement représenter sa composante connexe. La fonction group and sort effectue cette étape.

Ensuite, on s'intéresse à filtrer les couples (clés, liste de valeurs) et garde uniquement ceux pour lesquels la clé est supérieure au minimum des valeurs (fonction filter good tuple).

Enfin, l'algorithme s'arrête lorsque la valeur du Count New Pair est égale à 0. Cela signifie qu'on ne génère plus de nouveaux couple. Nous avons donc la garantie que chaque sommet est représenté par le plus petit sommet de sa composante connexe.

Cet algorithme a aussi pour particularité de supprimer pour chaque couple, la valeur minimale de la liste de valeurs. Le but étant de pouvoir appliquer une boucle commune sur le reste des valeurs sans prendre en compte le minimum. Nous nous sommes vite rendu compte que cette opération était inutile et qu'il suffisait simplement de commencer la boucle à partir du deuxième élément.

2.2 Aperçu de l'algorithme

```

1  def duplicate(edge):
2      return edge.map(lambda x : (x[1],x[0])).union(edge)
3
4  def group_and_sort(edge):
5      return edge.groupByKey().mapValues(list).mapValues(sorted)
6
7  def filter_good_tuple(edge):
8      return edge.filter(lambda x: x[0]>x[1][0])
9
10 def CCF_iteration(element):
11     liste=[]
12     cpt=False
13     cle=element[0]
14     min_liste=element[1][0]
15     del element[1][0]
16     for i in element[1] :
17         liste.append((i,min_liste))
18         cpt=True
19     liste.append((cle,min_liste))
20     return (liste,cpt)
21

```

```

21
22 def deroulement_CCF(edge):
23     cpt=1
24     while(cpt!=0):
25         edge=duplicate(edge)
26         edge=group_and_sort(edge)
27         edge=filter_good_tuple(edge)
28         edge=edge.map(lambda x: CCF_iteration(x))
29         cpt=sum(edge.map(lambda x:x[1]).collect())
30         edge=edge.flatMap(lambda x:x[0])
31         print(edge.collect())
32     return edge

```

2.3 Analyse des performances

Cet algorithme n'est pas très performant en termes de résultat. En prenant l'exemple de l'énoncé, on obtient une résolution de l'ordre des 20 secondes.

Néanmoins, nous avons choisi de l'insérer dans notre rapport car il a fait partie de notre processus de recherche de l'algorithme optimale.

Algorithme Final

3.1 Présentation

Voici présentée ci-dessous la version finale de l'algorithme. Il est plus efficace que les précédentes versions.

Il fonctionne de la manière suivante :

L'algorithme prend en entrée un RDD contenant un ensemble d'arrêtes, et pour chaque arrête (A,B) il génère l'arrête (B,A), de sorte que A soit dans la liste d'adjacence de B et vice versa (cela correspond à notre phase de Map).

Ensuite, il transforme le RDD en une liste afin de le parcourir plus facilement

Il utilise un compteur qui est initialisé à 0 et un *set()* qui va contenir tous les couples distincts qui seront générés à la fin de cette iteration. Notre choix s'est porté sur un set et pas sur une liste, car un set ne peut pas contenir de valeurs redondantes (le même couple ne peut pas apparaître deux fois). Ce qui nous permettra d'éviter un *distinct()* à la fin qui sera très coûteux en terme de calculs. Aussi les listes seront plus courtes ce qui permet un gain en espace de stockage.

Dans la phase Reduce, il parcourt le RDD préalablement transformé en liste, pour chaque couple (clé, liste d'adjacence) il calcule la valeur minimale de la liste d'adjacence. Cette dernière sera stockée dans la variable minimum.

Après cela, il filtre le RDD afin de ne garder que les couples (clé, liste d'adjacence) qui vérifient la condition $\text{clé} > \text{minimum}$. Ensuite les couples (clé, minimum) seront rajoutés au set.

Il parcourt la liste d'adjacence de chaque clé et pour chaque valeur différente de la variable minimum il rajoute au Set un couple (valeur, minimum) et incrémente le compteur.

À la fin de l'itération, le set est transformé en RDD et retourné avec le compteur.

3.2 Aperçu de l'algorithme

```

1  def ccf_iteration(edges):
2      #for each edge (A, B) it generates the edge (B, A), so that A is in the adjacency list of B and vice versa
3      rdd1 = edges.union(edges.map(lambda x : (x[1],x[0]))).groupByKey().mapValues(list)
4      rdd2 = rdd1.collect()
5      #a set () which will contain all the distinct pairs which will be generated at the end of this iteration
6      liste_values = set()
7      #counter which is initialized to 0
8      cpt = 0
9      for couple in rdd2 :
10         #the minimum value of the adjacency list
11         minimum = min(couple[1])
12         if couple[0]>minimum:
13             liste_values.add((couple[0],minimum))
14             for value in couple[1]:
15                 if(value != minimum):
16                     cpt = cpt+1
17                     liste_values.add((value,minimum))
18     edges_sortie = spark.sparkContext.parallelize(liste_values)
19
20     return edges_sortie,cpt

```

La fonction "result algo" présentée ci-dessous, appelle la fonction "ccf iteration" chaque fois que le compteur est différent de 0.

Cette fonction retourne le RDD final ainsi que le nombre d'itérations pour un graphe donné.

```

1  def result_algo(rdd):
2      cpt = 1
3      nombre_iteration=0
4      while(cpt!=0):
5          rdd,cpt = ccf_iteration(rdd)
6          nombre_iteration=nombre_iteration+1
7      return rdd,nombre_iteration

```

Remarque : Pour calculer le nombre de composantes connexes il suffit de prendre le rdd en sortie de la fonction "result algo" et de lui appliquer :

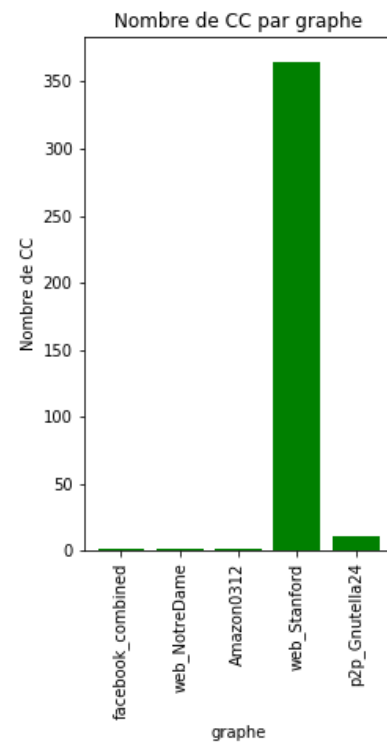
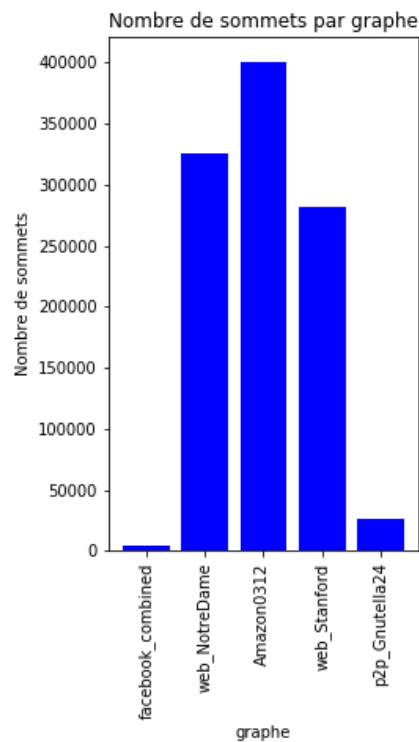
`rdd.map(lambda x : (x[1],x[0])).groupByKey().count()`

3.3 Analyse des performances

Dans cette phase d'analyse, nous avons testé l'algorithme final sur plusieurs graphes que nous avons récupérés sur le site *snap.stanford.edu/data/*.

Vous trouverez ci-dessous un tableau qui contient les caractéristiques de quelques graphes utilisés :

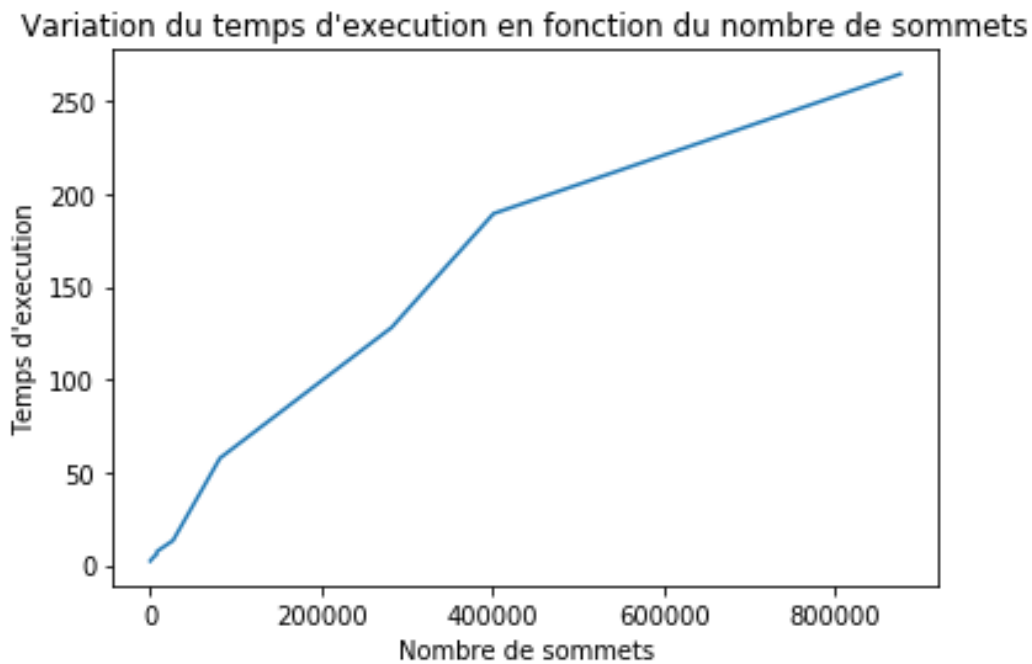
Nom du graphe	Nombre de sommet	Nombre de composante connexe
facebook-combined	4039	1
ca-HepTh	9877	427
roadNet-TX	1379917	424
web-Google	875517	2746
twitter-combined	81306	1
lastfm-asia-edges	7624	1
Wiki-Vote	7115	24
Exemple de l'article	8	2



3.3.1 Sur Databricks

Nous avons fait tourner notre algorithme sur les graphes présentés auparavant. L'objectif est de comparer le temps d'exécution et le nombre d'itérations en fonction du nombre de sommets de chaque graphe.

Ci-dessous une courbe montrant l'évolution du temps de calcul(en secondes) en fonction du nombre de sommets



Analyse de la courbe : le graphique montre corrélation entre le nombre de sommets contenus dans le graphe et le temps d'exécution de l'algorithme. Le temps d'exécution est donc proportionnel au nombre de sommets. Il est important de noter que la relation est linéaire et non exponentielle.

Remarque : Nous constatons que le nombre de composantes connexes peut influencer le temps d'exécution. Ainsi, pour deux graphes ayant quasiment le même nombre de sommets, nous remarquons que celui ayant plus de composantes connexes prend légèrement moins de temps que l'autre. Notre hypothèse est de dire que lorsqu'un graphe possède un nombre important de composantes connexes, cela implique qu'il possède forcément moins d'arcs ou d'arêtes, donc moins de calculs à effectuer par l'algorithme. Prenons l'exemple des deux fichiers *act mooc* et *lastfm-asia-edges*

Nom	Nombre de sommets	Nombre de CC	Temps
lastfm-asia-edges	7624	1	6.3180 s
act mooc	7606	24	5.3480 s

Ci-dessous un tableau montrant le temps d'exécution ainsi que le nombre d'itération pour chaque graphe

Nom du graphe	temps d'exécution	Nombre d'iteration
facebook-combined	4.5466 s	5
ca-HepTh	7.990 s	7
roadNet-TX	911,569 s	14
twitter-combined	57.568 s	6
lastfm-asia-edges	7.230 s	7
Wiki-Vote	5.341 s	5
Exemple de l'article	1.987 s	2
web-NotreDame	75.3465 s	7
Amazon0312	189.2931 s	7
web-Stanford	128.018 s	13
p2p-Gnutella24	13.223 s	6

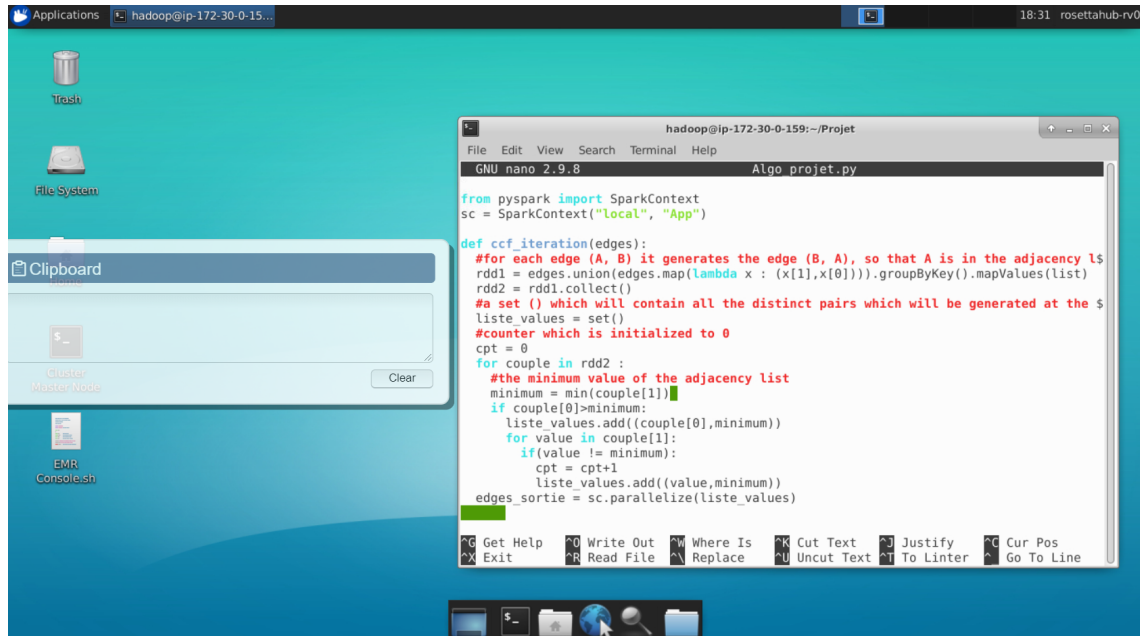
Analyse des performances de l'algorithme sur le graphe *web-Google*

Pour optimiser nos performances nous avons choisi ce graphe comme référence car nous avons un temps d'exécution de cible à atteindre, celui cité dans l'article. Le nombre d'itérations est resté stable mais nous avons amélioré quotidiennement notre temps d'exécution. Nous ne montrons pas ici les premières versions de l'algorithme car trop lentes, mais à partir de l'algorithme transitoire, les performances deviennent mesurables. Enfin, notre dernière version arrive à dépasser le temps d'exécution de l'algorithme cité dans l'article. Notre meilleure performance est de 8 itérations en 240 secondes.

Algorithme	temps d'exécution	Nombre d'iteration
Algorithme transitoire	1256 s	8
Algorithme final	240 s	8
Algorithme utilisé dans l'article	256 s	11

3.3.2 Sur RosettaHub

Nous avons testé notre algorithme sur RosettaHub, ci-dessous les captures d'écran décrivant son exécution.



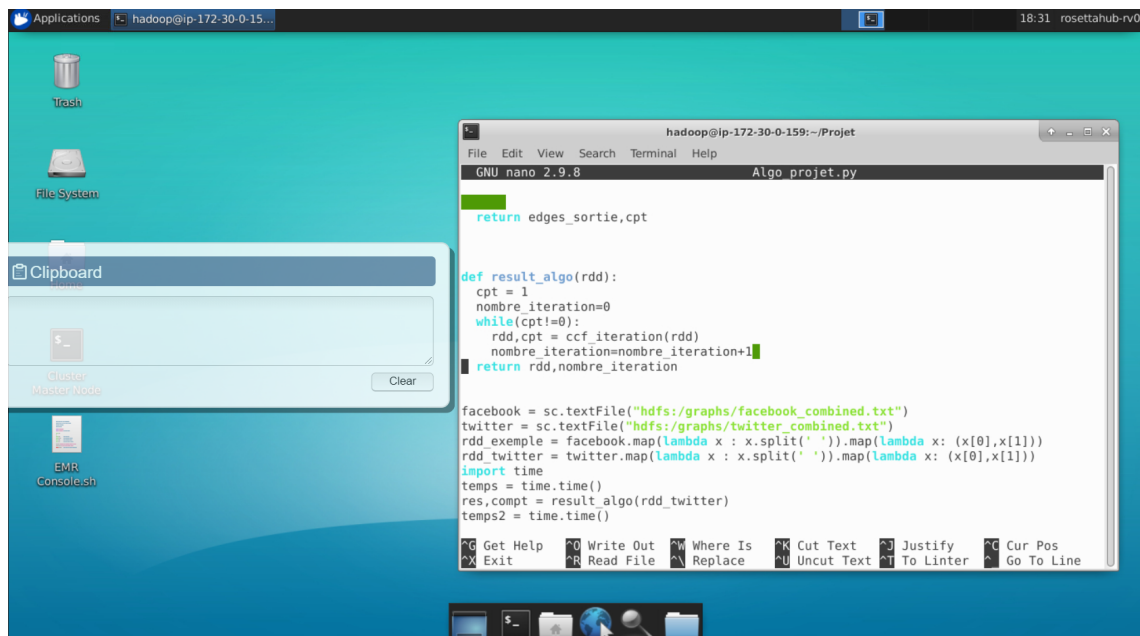
```

hadoop@ip-172-30-0-159:~/Projet
GNU nano 2.9.8 Algo projet.py

from pyspark import SparkContext
sc = SparkContext("local", "App")

def ccf_iteration(edges):
    #for each edge (A, B) it generates the edge (B, A), so that A is in the adjacency list
    rdd1 = edges.union(edges.map(lambda x : (x[1],x[0])))
    rdd2 = rdd1.collect()
    #a set () which will contain all the distinct pairs which will be generated at the
    liste_values = set()
    #counter which is initialized to 0
    cpt = 0
    for couple in rdd2 :
        #the minimum value of the adjacency list
        minimum = min(couple[1])
        if couple[0]>minimum:
            liste_values.add((couple[0],minimum))
        for value in couple[1]:
            if(value != minimum):
                cpt = cpt+1
                liste_values.add((value,minimum))
    edges_sortie = sc.parallelize(liste_values)

```



```

hadoop@ip-172-30-0-159:~/Projet
GNU nano 2.9.8 Algo projet.py

return edges_sortie,cpt

def result_algo(rdd):
    cpt = 1
    nombre_iteration=0
    while(cpt!=0):
        rdd,cpt = ccf_iteration(rdd)
        nombre_iteration=nombre_iteration+1
    return rdd,nombre_iteration

facebook = sc.textFile("hdfs://graphs/facebook_combined.txt")
twitter = sc.textFile("hdfs://graphs/twitter_combined.txt")
rdd_exemple = facebook.map(lambda x : x.split(' ')).map(lambda x: (x[0],x[1]))
rdd_twitter = twitter.map(lambda x : x.split(' ')).map(lambda x: (x[0],x[1]))
import time
temps = time.time()
res,cpt = result_algo(rdd_twitter)
temps2 = time.time()

```

Voici maintenant des captures d'écran représentant des résultats de test. Nous avons testé notre algorithme sur les graphes de relations Twitter et Facebook implémentés précédemment avec Databricks.

```

hadoop@ip-172-30-0-159:~/Projet
File Edit View Search Terminal Help
ompleted, from pool
21/02/10 18:29:21 INFO DAGScheduler: ResultStage 11 (collect at /home/hadoop/Projet/Al
go_projet.py:7) finished in 0.763 s
21/02/10 18:29:21 INFO DAGScheduler: Job 5 finished: collect at /home/hadoop/Projet/Al
go_projet.py:7, took 1.921134 s
21/02/10 18:29:21 INFO BlockManagerInfo: Removed taskresult_23 on ip-172-30-0-159.eu-w
est-1.compute.internal:45571 in memory (size: 1509.7 KB, free: 1038.7 MB)
temps de calcul : 61.37920045852661
21/02/10 18:29:21 INFO SparkContext: Invoking stop() from shutdown hook
21/02/10 18:29:21 INFO SparkUI: Stopped Spark web UI at http://ip-172-30-0-159.eu-west
-1.compute.internal:4040
21/02/10 18:29:21 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint
stopped!
21/02/10 18:29:21 INFO MemoryStore: MemoryStore cleared
21/02/10 18:29:21 INFO BlockManager: BlockManager stopped
21/02/10 18:29:21 INFO BlockManagerMaster: BlockManagerMaster stopped
21/02/10 18:29:21 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: Output
CommitCoordinator stopped!
21/02/10 18:29:21 INFO SparkContext: Successfully stopped SparkContext
21/02/10 18:29:21 INFO ShutdownHookManager: Shutdown hook called
21/02/10 18:29:21 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-90ea06d7
-15e1-4a0d-90ef-dac6ba9ed8f2/pyspark-5a1ad5b2-baad-420a-914e-6d85edc97307
21/02/10 18:29:21 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-90ea06d7
-15e1-4a0d-90ef-dac6ba9ed8f2
21/02/10 18:29:21 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-d761cbl1a
-d953-4782-8e9e-35e30ce2ef4
[hadoop@ip-172-30-0-159 Projet]$

```

```

hadoop@ip-172-30-0-159:~/Projet
File Edit View Search Terminal Help
mpleted, from pool
21/02/10 18:10:01 INFO DAGScheduler: ResultStage 9 (collect at /home/hadoop/Projet/Alg
o_projet.py:7) finished in 0.130 s
21/02/10 18:10:01 INFO DAGScheduler: Job 4 finished: collect at /home/hadoop/Projet/Al
go_projet.py:7, took 0.296239 s
temps de calcul : 6.027446269989014
21/02/10 18:10:01 INFO SparkContext: Invoking stop() from shutdown hook
21/02/10 18:10:02 INFO SparkUI: Stopped Spark web UI at http://ip-172-30-0-159.eu-west
-1.compute.internal:4040
21/02/10 18:10:02 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint
stopped!
21/02/10 18:10:02 INFO MemoryStore: MemoryStore cleared
21/02/10 18:10:02 INFO BlockManager: BlockManager stopped
21/02/10 18:10:02 INFO BlockManagerMaster: BlockManagerMaster stopped
21/02/10 18:10:02 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: Output
CommitCoordinator stopped!
21/02/10 18:10:02 INFO SparkContext: Successfully stopped SparkContext
21/02/10 18:10:02 INFO ShutdownHookManager: Shutdown hook called
21/02/10 18:10:02 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-f7eaaac8
-e8b8-48ea-b50a-8602cce18a3c/pyspark-ba2faf33-b70c-4e5d-b293-fcd32886c7c3
21/02/10 18:10:02 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-612efe7f
-3847-460e-8083-bfd4a024980b
21/02/10 18:10:02 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-f7eaaac8
-e8b8-48ea-b50a-8602cce18a3c
[hadoop@ip-172-30-0-159 Projet]$ ls
Algo_projet.py facebook_combined.txt
[hadoop@ip-172-30-0-159 Projet]$

```

Concernant les résultats obtenus :

- Pour Twitter on obtient un résultat de 61 secondes
- Pour Facebook, on obtient un résultat de 6.01 secondes

En comparaison, avec Databricks on obtenait comme résultat :

- Pour Twitter 57 secondes
- Pour Facebook 4,54 secondes

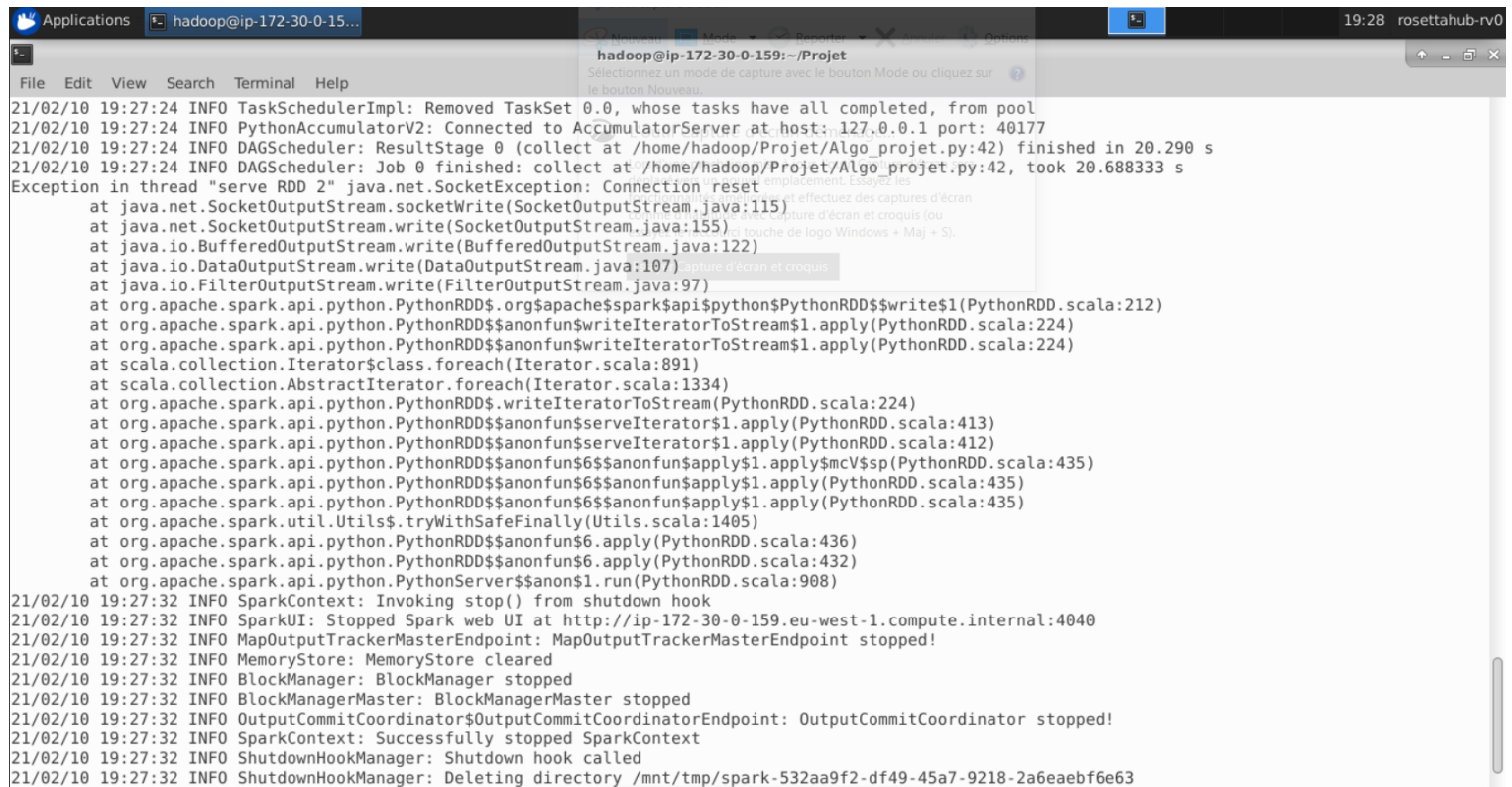
Ainsi, le choix d'utilisation de RosettaHub pour des graphes similaires à ceux-ci n'est que peu pertinent. Car en effet la vitesse d'exécution de l'algorithme est plus lente que sur Databricks et l'implémentation est plus complexe.

Conclusion

Pour conclure cette analyse, nous pouvons dire que nous avons réussi à implémenter un algorithme fonctionnel et efficace qui permet de résoudre le problème de recherche des composantes connexes dans un graphe. Néanmoins, l'utilisation de cet algorithme sur des graphes volumineux soulève d'autres problématiques liées à la gestion des systèmes distribués et à la gestion de la mémoire. Un exemple des problématiques rencontrées à ce sujet se trouve en annexe. L'utilisation de Rosetta ne nous a pas aidé à obtenir des résultats de calcul meilleurs sur des graphes de taille moyenne et nous a causé des erreurs d'exécution pour des graphes volumineux. Nous n'avons donc malheureusement pas expérimenté l'efficacité de cet outil.

Annexe

Exemple d'erreur obtenue lors de l'exécution de notre algorithme sur Rosetta sur un graphe possédant un grand nombre de sommets



```
Applications  hadoop@ip-172-30-0-15... 19:28 rosettahub-rv0
hadoop@ip-172-30-0-159:~/Projet
File Edit View Search Terminal Help
21/02/10 19:27:24 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
21/02/10 19:27:24 INFO PythonAccumulatorV2: Connected to AccumulatorServer at host: 127.0.0.1 port: 40177
21/02/10 19:27:24 INFO DAGScheduler: ResultStage 0 (collect at /home/hadoop/Projet/Algo_projet.py:42) finished in 20.290 s
21/02/10 19:27:24 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/Projet/Algo_projet.py:42, took 20.688333 s
Exception in thread "serve RDD 2" java.net.SocketException: Connection reset
    at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:115)
    at java.net.SocketOutputStream.write(SocketOutputStream.java:155)
    at java.io.BufferedOutputStream.write(BufferedOutputStream.java:122)
    at java.io.DataOutputStream.write(DataOutputStream.java:107)
    at java.io.FilterOutputStream.write(FilterOutputStream.java:97)
    at org.apache.spark.api.python.PythonRDD$.org$apache$spark$api$python$PythonRDD$$write$1(PythonRDD.scala:212)
    at org.apache.spark.api.python.PythonRDD$$anonfun$writeIteratorToStream$1.apply(PythonRDD.scala:224)
    at org.apache.spark.api.python.PythonRDD$$anonfun$writeIteratorToStream$1.apply(PythonRDD.scala:224)
    at scala.collection.Iterator$class.foreach(Iterator.scala:891)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1334)
    at org.apache.spark.api.python.PythonRDD$.writeIteratorToStream(PythonRDD.scala:224)
    at org.apache.spark.api.python.PythonRDD$$anonfun$serveIterator$1.apply(PythonRDD.scala:413)
    at org.apache.spark.api.python.PythonRDD$$anonfun$serveIterator$1.apply(PythonRDD.scala:412)
    at org.apache.spark.api.python.PythonRDD$$anonfun$6$$anonfun$apply$1.apply$mcV$sp(PythonRDD.scala:435)
    at org.apache.spark.api.python.PythonRDD$$anonfun$6$$anonfun$apply$1.apply(PythonRDD.scala:435)
    at org.apache.spark.api.python.PythonRDD$$anonfun$6$$anonfun$apply$1.apply(PythonRDD.scala:435)
    at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1405)
    at org.apache.spark.api.python.PythonRDD$$anonfun$6.apply(PythonRDD.scala:436)
    at org.apache.spark.api.python.PythonRDD$$anonfun$6.apply(PythonRDD.scala:432)
    at org.apache.spark.api.python.PythonServer$$anon$1.run(PythonRDD.scala:908)
21/02/10 19:27:32 INFO SparkContext: Invoking stop() from shutdown hook
21/02/10 19:27:32 INFO SparkUI: Stopped Spark web UI at http://ip-172-30-0-159.eu-west-1.compute.internal:4040
21/02/10 19:27:32 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
21/02/10 19:27:32 INFO MemoryStore: MemoryStore cleared
21/02/10 19:27:32 INFO BlockManager: BlockManager stopped
21/02/10 19:27:32 INFO BlockManagerMaster: BlockManagerMaster stopped
21/02/10 19:27:32 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
21/02/10 19:27:32 INFO SparkContext: Successfully stopped SparkContext
21/02/10 19:27:32 INFO ShutdownHookManager: Shutdown hook called
21/02/10 19:27:32 INFO ShutdownHookManager: Deleting directory /mnt/tmp/spark-532aa9f2-df49-45a7-9218-2a6eae6f6e63
```