Analyse du besoin :

Pour bien comprendre les fonctionnalités globales des systèmes, ces derniers sont souvent modélisés sous formes de graphes.
Ces systèmes sont en effet très nombreux, il peut s'agir de protéines, de composés chimiques, mais également d'internet ou des réseaux sociaux tel que Facebook ou Instagram ...

Un graphe est un réseaux qui relie plusieurs objets que l'on nomme des nœuds par des arcs, il existe de nombreuses techniques d'extractions de graphes qui ont été développées pour extraire efficacement de l'information, mais également pour faire des analyses sur les différentes caractéristiques de ces réseaux complexes que l'on appelle: les algorithme pour la théorie des graphes (algorithme de parcours, algorithme le plus court chemins tel que **Dijkstra** et l'algorithme de **Belleman-ford** et de nombreux d'autres ….),

En effet, on distingue deux types de graphes, il y'a d'une part les **graphes orienté** et d'autre part les **graphes non orientés**

Le graphe non orienté peut contenir un ou plusieurs composantes connexes, un composant connexe est un ensemble de nœuds qui sont reliés deux à deux par au moins un chemin, notre objectif dans ce projet est d'implémenter un algorithme qui permet de retrouver le nombre de composantes connexes d'un graphe à grande échelle comme celui de Facebook, twitter … etc.
Cet algorithme permet de trouver le graphe connexe des amis sur les réseaux sociaux, pour pouvoir par la suite suggérer l'amitié pour les autres utilisateurs qui appartiennent au même composant connexe

Partie 1 :

Algorithme CCF Itérateur : (Connected Component Finder) cet algorithme MapReduce qui permet de retrouver les composantes connexes du graphe, l'entrée de cet algorithme est une liste de couple qui représente la liste des arêtes qui relient les nœuds dans le graphe, et la sortie de l'algorithme est une liste de couples qui relie la valeur minimale de chaque composante connexe du graphe avec chaque nœud de ce dernier (min valeur, valeur nœud i).

Au début, le job Map transforme les couples reçus en entrée en un couple (key, value) tel que key représente un nœud quelconque du graphe, value représente la liste des nœuds adjacents à ce nœud, c'est les nœuds qui sont connectés avec le nœud key directement par une arête, après on a le jobe Redue qui prend en entrée le couple (vid nœud , liste des Adjacents) et pour chaque nœud de la liste renvoie (de V1, …..,Vn) si Vid est plus grand que le nœud minimum dans la liste des nœuds adjacents, on retourne d'abord (Vid, Vimin) , puis pour chaque Vi en retourne le couple (Vi,Vimin) tel que Vid est différent de Vimin, et à chaque fois qu'on a un couple différent, on ajoute à la variable Counter +1 (on initialise la variable Counter à zéro au début de l'algorithme (c'est une variable globale)), mais dans le cas ou Vid est plus petit que Vimin on renvoie aucun couple , et on répète ces deux jobs plusieurs fois jusqu'à que la valeur du Counter soit égales à zéro.

ALGORITHME CCF-Dedup : Durant l'exécution de l'algorithme CCF-iterate le même couple peut être retourner plusieurs fois, c'est pour cela qu'on a implémenter l'algorithme CCF-Dedup qui permet d'enlever la duplication , cette version d'algorithme est plus efficace que la première en termes de vitesse d'exécution.

voilà un exemple d'exécution de l'algorithme CCF-itérateur sur la figure1, et comme c'est la dernière itération de l'algorithme, on pourra bien remarquer que l'algorithme renvoie deux composants connexes:
composant 1 : contient les couples [(B,A),(C,A),(D,A),(D,A),(E,A)],
composant 2: [(G,F),(H,F)]

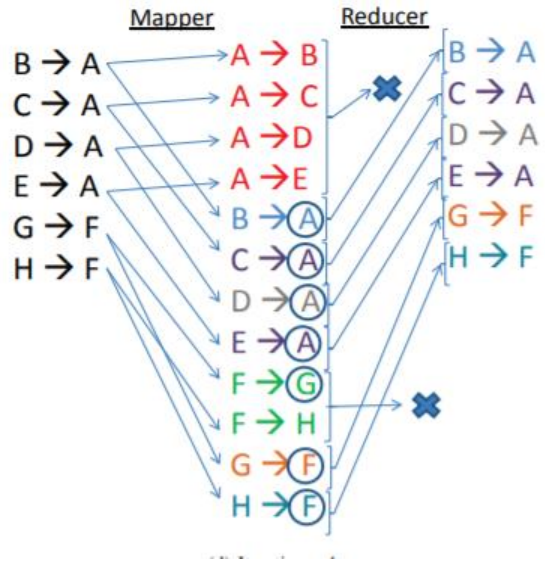


Figure 1

Description du code pyspark:

partie 1: Algorithme CCF itérateur:

on commence par créer la variable itérations et on l' initialise à zéro pour pouvoir compter le nombre d'itérations de l'algorithme par la suite

```
itérations=0
```

puis on charge le le contenu du fichier qui contient les arêtes du graphe sous form des couples dans une variable qu'on appelle "**graphe**" via la ligne du code suivante :

```
graphe=sc.textFile("/home/hossem/Bureau/Bigdata/pythonProject/large_graphes_connexes_component/testFiles/twitter_combined.txt")
```

puis on supprime les espaces entre chaque couple ligne par ligne via la ligne du code suivante:

```
sortie = graphe.map(lambda x : x.split(' '))
```

pour compter à chaque itération le nouveau couple produit par l'algorithme on utilise la variable acc (accumulateur) qu'on initialise à zéro au début :

```
acc = sc.accumulator(0)
```

puis on initialise la variable stop qui sert à stopper les itérations quand elle sera égale à zéro, on initialise la variable à 1 au début comme suit :

```
stop =1
```

on initialise la variable **tps_EUR** à 1 par un trimer pour pouvoir compter le temps d'exécution de l'algorithme :

```
tps_EUR1 = time.time()
```

et maintenant tant que la variable **"stop**" est différente de zéro :

```
while stop != 0:
```

on réinitialise à chaque fois l'accumulateur à zéro

```
acc.value = 0
```

on utilise la méthode **flatMap()** pour inverser les couples et les mettre dans même RDD, exemple si on a en entrée le couple (A , B) en aura dans le rdd **graphe_ccf** =((A,B),(B,A)) :

```
graphe_ccf = sortie.flatMap(lambda x: [(x[0], x[1]), (x[1], x[0])]).
```

par la suite on utilise la méthode **groupeByKey()** pour grouper les couple par clé, puis on utilise la méthode **map()** pour mettre la sortie sous forme de couple (vi, [liste de voisins]) quand met dans la variable **graphe_ccf** :

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

ensuite on applique la méthode traitement définis ci-dessous , en utilisant la méthode **flatMap()** de pyspark qui fait un certain traitement sur chaque couple du rdd **graphe_ccf**.

```
sortie = graphe_ccf.flatMap(lambda x: traitement(x[0], x[1]))
```

on met le résultat de la sortie dans la variable résultat en utilisant le méthode **collect()** qui permet de récupérer tous les éléments de l'ensemble de données.

```
resultat = sortie.collect()
```

et à chaque fois en mets a jours le la variable stop par la valeur de l'accumulateur.

```
stop = acc.value
```

la fin de ma boucle while .

Ce traitement de la boucle while se répète jusqu' à que la valeur stop devient nulle.

on enregistre le temps une deuxième fois dans la variable **tps_EUR2** :

```
tps_EUR2 = time.time()
```

puis en fait la différence entre les deux valeur et on multiplie fois mille pour avoir le résultat en second :

```
executionTime = ((tps_EUR2-tps_EUR1)*1000)
```

on affiche le temps d'exécution :

```
print(executionTime)
```

on affiche le résultat de l'algorithme :

```
print(resultat)
```

Méthode Traitement :

```
def traitement(key,values):

      minval = min(values)

      result=[]

      if minval<key:

      result.append((key,minval))

      for value in values:

            if(minval != value):

                  acc.add(1)

                  result.append((value,minval))



      return result
```

cette méthode prend en entrée (key, value), tel que key : représente l'identifiant d'un nœud du graphe, et le deuxième paramètre représente la liste des nœuds voisins de ce nœud **key**, au début on cherche la valeur minimale dans la liste des adjacents et on l'ajoute dans la variable **minVal,** on initialise la liste des resultat par une liste vide. et si la **minVal** est inférieur à la valeur du **nœud key,** on ajoute au résultat le couple (key , minVal).

Ensuite on parcourt la listes des nœuds adjacents et si la valeur du nœud est différente de minVal, on ajoute dans la variable **résultat** le couple (value i, minVal)

2-Algorithme CCF-sorted :

Au début, on charge le fichier de données dans la variable graphe comme suite:

```
graphe=sc.textFile("/home/ahmed/Bureau/Bigdata/pythonProject/large_graphes_connexes_component/testFiles/web-Google.txt")
```

on supprime les espaces par la méthode **map()** en utilisant la fonction **split(' '):**

```
entre=graphe.map(lambda x : x.split(' '))
```

puis on initialise l'accumulateur a 1 qui vas nous permettre de compter le nombre des nouveaux couples produite par l'algorithme :

```
acc = sc.accumulator(1)
```

on initialise aussi la variable "stop" à 1 aussi, cette variable sert à sortir de la boucle while quand sa valeur sera égale à zéro (elle a les mêmes valeur que l'accumulateur) :

```
stop=1
```

et tant que la valeur “stop” est différent de zéro:

```
while stop !=0:
```

on initialise l’accumulateur à zéro :

```
acc.value = 0
```

ensuite on inverse les couples reçu en entrée par la méthode **flatMap()** et on les ajoute dans la variable **grapheCcf** , exemple si le variable entrée contient (A , C) et (B,A) => grapheCcf= ((A,C)),(C,A), (B,A),(A,B))

```
grapheCcf = entre.flatMap(lambda x :
[(x[0],x[1]),(x[1],x[0])])
```

puis on regroupe ces couples via la méthodes **groupeByKey(),** sur l’exemple précédent la méthode **groupeByKey()** nous retourne les couples suivants (A,[C,A,B]) (B,A) sans appliquer la méthodes **sorted(),** la méthode sorted sert à trier la liste des nœuds par ordre croissant comme suit: (A,sorted[C,A,B]) => (A,[A,B,C])

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

ensuite on applique on applique la méthode **traitementCcfWithSecondarySorting** définis ci-dessous en utilisant la méthodes **flatMap()** de pyspark comme suit:

```
iterate = grapheCcf.flatMap(lambda x:
traitementCcfWIthSecondarySorting(x[0], x[1]))
```

par la suite on supprime les doublons en utilisant la ligne de code suivante

```
edup=iterate.map(lambda x:((x[0],x[1]),0)).reduceByKey(lambda
x,y:x+y).flatMap(lambda x: [(x[0][0],x[0][1])] if x[0][1]!=x[0][0]
else [])
```

on met à jours la valeur du rdd **entrée** avec la valeur du **dedup** et on refait le même traitement jusqu'à ce que la valeur de l’accumulateur soit égales à zéro

```
entre=dedup
```

on met le résultat de la variable **dedup** dans la variable **résultat** en utilisant le méthode **collect()** qui permet de récupérer tous les éléments de l’ensemble de données.

```
result = dedup.collect()
```

on met a jours la valeur de la variable stop :

```
stop=acc.value
```

fin de la boucle while

on affiche le résultat de l’algorithme :

```
print(result)
```

Méthode traitementCcfWithSecondarySorting :

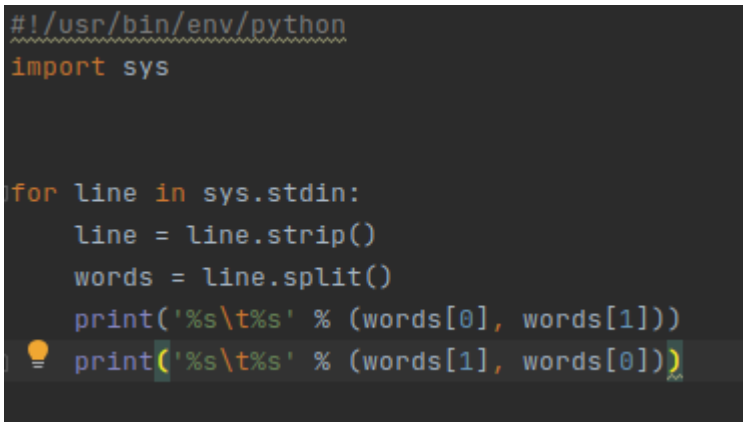
cette méthodes prend en entrée un couple (Key, values) tel que, **key** représente la valeur d’un nœud dans le graphe, values : est la liste des nœuds triés par la méthode sorted vu précédemment, maintenant pour avoir la valeur minimale de la liste on prend juste la première valeur de la liste c.-a-d. values[0] car la liste est triée, ce traitement nous a permis d’optimiser l’algorithme CCF car on supprimer la coût de rechercher la valeur minimale de la liste à chaque itération et ça nous fait un gain énorme en terme du temps d'exécution

```
def traitementCcfWIthSecondarySorting(key, values):

      minVal = values[0]

      resultat= []

      if minVal < key:

      resultat.append((key,minVal))

      for value in values:

            if(minVal != value):

                  resultat.append((value, minVal))

                  acc.add(1)



      return resultat
```

Partie 4 : JOB MapReduce

Dans cette partie nous avons implémenté l'algorithme CCF Iterate en Job MapReduce. Ensuite nous avons testé cette implémentation en local sur linux Ubuntu.

Mapper :



```
#!/usr/bin/env/python
import sys


for line in sys.stdin:
    line = line.strip()
    words = line.split()
    print('%s\t%s' % (words[0], words[1]))
    print('%s\t%s' % (words[1], words[0]))
```

Le reducer :

```python
import sys

currentKey = None
tmplist = []
conteur = 0

def CcfIteration(key, values):
    cpt = 0
    minVal = key
    valueList = []
    elements = values
    for elm in elements:
        if elm < minVal:
            minVal = elm
        valueList.append(elm)
    if minVal < key:
        print('%s\t%s' % (key, minVal))
        for val in valueList:
            if minVal != val:
                cpt += 1
                print('%s\t%s' % (val, minVal))
    return cpt
```

```python
for line in sys.stdin:
    line = line.strip()
    currentLine = line.split('\t')
    key = currentLine[0]
    value = currentLine[1]
    if currentKey is None:
        currentKey = key
        tmplist.append(value)
    else:
        if currentKey == key:
            tmplist.append(value)
        else:
            conteur += CcfIteration(currentKey, tmplist)
            currentKey = key
            tmplist = [value]
if key == currentKey:
    conteur += CcfIteration(currentKey, tmplist)
print(conteur)
```

Dans ce reducer nous avons fait exprès de ne pas utiliser la mémoire en utilisant un dictionnaire ou une liste mais en utilisant directement le buffer et en prenant en compte que Hadoop renvoie les clés triées après la phase de Shuffle&Sort.

partie 5 : implémentation du payspark dans Hadoop

Cette partie résume l'implémentation des algorithmes sur une machine virtuelle AWS hébergée sur le cloud Rosetta. Nous avons eu accès à cette plateforme avec un crédit limité pour créer une machine virtuelle avec Hadoop afin d'exécuter les algorithmes sur des données massives et des graphes larges. Sur l'espace Workspace on a réussi à créer notre machine virtuelle. La méthode est comme suit :

- cliquez sur l'onglet Workspace
- cliquez sur l'onglet Big data LAB
- choisissez la machine v3 - Data Science for Business - Hadoop, Spark
- attendez 10 min afin que la machine démarre

Cette machine comporte 6 clusters avec une mémoire de 2GB et une mémoire totale de 6GB pour le cluster maître.

La première étape consiste à trouver comment fonctionne notre code pyspark sur la version Hadoop de la machine. Tout d'abord on commence par importer les fichier de données. Pour cette partie on a 3 fichier de données:

- facebook.txt : un fichier contenant 4,039 nœud et 88,234 arrêts
- twitter.txt : un fichier contenant 81,306 nœud et 1,768,149 arrêts
- web-google.txt : un fichier contenant 875,713 nœuds et 5,105,039 Arrêts

Ces fichiers se trouvent dans le site https://snap.stanford.edu/data/ .
On a choisi ces 3 fichiers pour conclure à petite, moyenne et grande échelle l'efficacité de notre algorithme. vu que les 3 fichiers sont de tailles différentes (petite, moyenne, grande).

Importer ces fichiers de data sur Hadoop :

Après une vérification de compatibilité de ces fichiers de données avec notre code, la prochaine étape est d'importer ces données dans notre machine virtuelle contenant Hadoop.

Pour importer les fichiers la méthode la plus simple est de télécharger ces fichiers dans un drive, récupérer leur liens et les télécharger sur Hadoop en utilisant la méthode wget. Ensuite, les renommer pour faciliter l'utilisation par les algorithmes. Enfin, nous les avons déplacés dans le répertoire principal Hadoop de la machine qui se trouve dans "/user/Hadoop/" avec la commande "hdfs dfs -put nomfichier /user/hadoop/".

Exemple d'un fichier :

```
wget
https://drive.google.com/u/0/uc?id=1euPO8FMBEMAUyzYilRGkxFAzKPU2t_JB
mv uc?id=1euPO8FMBEMAUyzYilRGkxFAzKPU2t_JB twitter.txt
hdfs dfs -put twitter.txt /user/hadoop/
```

2 . Importer les sources:

Le 2e étape consiste à importer les fichiers d'implémentation des algorithmes ccf iterates, et ccf iterate with sorting dans notre machine virtuelle afin de les exécuter à l'aide de pyspark. Comme le code source des méthode est relativement court, la méthode la plus simple est de créer ces fichiers dans l'EMR et copier le code source à l'intérieur.

Example :

```
touch test.py
nano test.py
< coller le code source et sauvegarder >
```

3. lancer les algorithmes :

Avant de lancer les sources il faut ajouter dans les sources l'objet sc de configuration afin que le pyspark puisse l'exécuter. Le code à ajouter est :

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("pyspark")
sc = SparkContext(conf=conf)
```

```
solution = sc.parallelize([result])
solution.saveAsTextFile("sortieCcfIterateSorting")
```

Cette partie du code sert à enregistrer le résultat de l'algorithme dans un fichier de sortie stocké dans Hadoop dans le répertoire "/user/hadoop/". Il faut préciser dans le code source quel fichier on doit traiter. Le code source est modifié avec 'nano source.py' et ensuite, enregistré.

Pour lancer le programme il suffit de saisir la ligne suivante :

```
spark-submit programme.py
```

La commande ‘spark-submit’ sert à exécuter un programme pyspark, cependant cette commande contient beaucoup de paramètres à préciser. Si les paramètres ne sont pas renseignés, la commande prend des paramètres par défaut.
parmi ces paramètres on peut citer :
`--master yarn`
préciser quel est le cluster maître par exemple yarn, ou local.
`--deploy-mode cluster`
pour préciser que le mode de déploiement est en cluster.

`--num-executors 5 --driver-memory 2g --executor-memory 2g --executor-cores 1`
Ces paramètres précisent le nombre, la capacité mémoire et le nombre de cœurs des ‘cluster slave’. Ainsi, la mémoire du driver.

Selon l'organisation voulue, ces paramètres sont modifiés en respectant la capacité maximum du cluster et la masse de données à traiter. Ainsi, notre temps d'exécution et nos traitements seront plus rapides si on alloue plus de mémoire.

Exécution des programmes :
Les programmes sont exécutés en 2 modes :
- ressources minimum (par défaut)
- ressources maximum.

les commandes pour exécuter les programmes avec les 2 modes sont :
Ressources minimums : `spark-submit ccf.py`

Ressources maximum :`spark-submit --master yarn --deploy-mode cluster --num-executors 6 --driver-memory 5500M --executor-memory 1900M --executor-cores 1 ccf.py`

Ensuite si l’exécution est réussie les résultats de l'exécution se trouve dans les partie de la sortie de l’algorithme dans le fichier de sortie ‘/user/hadoop/sortieCcfIterate/’. l’affichage de toutes les parties se fait avec la commande suivantes

‘Hdfs dfs -cat /user/hadoop/sortieCcfIterate/part*’

```
[hadoop@ip-172-30-2-61 ~]$ hdfs dfs -cat /user/hadoop/sortieCcfIterate/part*
4038
[('1', '0'), ('48', '0'), ('53', '0'), ('54', '0'), ('73', '0'), ('88', '0'), ('119', '0'), ('280', '0'), ('299', '0'), ('315', '0'), ('346', '0'), (
'4', '0'), ('328', '0'), ('8', '0'), ('91', '0'), ('110', '0'), ('259', '0'), ('264', '0'), ('9', '0'), ('21', '0'), ('26', '0'), ('56', '0'), ('66',
 '0'), ('69', '0'), ('113', '0'), ('122', '0'), ('128', '0'), ('134', '0'), ('141', '0'), ('161', '0'), ('185', '0'), ('188', '0'), ('231', '0'), ('2
52', '0'), ('258', '0'), ('272', '0'), ('277', '0'), ('285', '0'), ('304', '0'), ('334', '0'), ('341', '0'), ('342', '0'), ('10', '0'), ('12', '0'),
('14', '0'), ('20', '0'), ('115', '0'), ('116', '0'), ('144', '0'), ('214', '0'), ('226', '0'), ('326', '0'), ('16', '0'), ('29', '0'), ('82', '0'),
('331', '0'), ('17', '0'), ('19', '0'), ('111', '0'), ('140', '0'), ('112', '0'), ('138', '0'), ('174', '0'), ('175', '0'), ('227', '0'), ('319', '0'
 ('44', '0'), ('162', '0'), ('333', '0'), ('40', '0'), ('98', '0'), ('108', '0'), ('121', '0'), ('184', '0'), ('223', '0'), ('248', '0'), ('274', '
```

Algorithme Ccf iterate 1 :

Sur des petites quantités de données:

- Avec un minimum de ressources, exécution du programme en ressources minimum sur le petit fichier de data facebook.txt est réussi et elle a pris 33 seconde d'exécution

- Avec un maximum de ressources, l'exécution de ce programme a pris 35 secondes et nous a généré 4038 composantes connexes .

Sur des quantités moyennes de données:

- En allouant un minimum de ressources, le fichier twitter.txt ne s'est pas exécuté et l'algorithme déclenche une exception liée à l'insuffisance des ressources.

- En allouant un maximum de ressources, l'algorithme s'exécute en 120 secondes. Cet algorithme trouve 81305 composantes connexes.

Sur des grandes quantités de données:

- En allouant un minimum de ressources : Le programme ne s'est pas exécuté avec des ressources minimum.
- En allouant un maximum de ressources : Le programme s'est exécuté en 700 secondes. en générant composantes connexes.

Algorithme 2: CCF_ Iterate with sorting :

Les mêmes textes ont été faits avec les mêmes fichiers de données. et les résultats pour cette approche sont les suivant:

petites quantités de données (facebook.txt) :

- En allouant un minimum de ressources, le programme s'est exécuté en 35 secondes, en générant 4038 composantes connexes.
- En Allouant un maximum de ressources, le programme s'est exécuté en 37 secondes, en générant 4038 composantes connexes.

moyennes quantités de données: (twitter.txt)

- En allouant un minimum de ressources, le fichier twitter.txt s'est exécuté en 100 secondes et générant 81305 composantes connexes.

- En allouant un maximum de ressources, le programme ne s'est pas exécuté. Un problème de mémoire est survenu.

Sur des quantités de données importantes:

- En allouant un minimum de ressources : Le programme ne s'est pas exécuté avec des ressources minimum.
- En allouant un maximum de ressources : le programme s'est exécuté en 378 secondes.

Test de l'algorithme sur sur data bricks:

Le site du Data bricks exécute l'algorithme mais en temps plus important. nous avons testé l'algorithme CCf with sorting sur data bricks et nous avons eu le résultat suivant:

```
#graphe=sc.textFile("dbfs:/FileStore/shared_uploads/mohamed-houssem.hamroun@dauphine.eu/facebook_combined.txt")
#graphe=sc.textFile("dbfs:/FileStore/shared_uploads/mohamed-houssem.hamroun@dauphine.eu/twitter_combined__1_.txt")
graphe=sc.textFile("dbfs:/FileStore/shared_uploads/mohamed-houssem.hamroun@dauphine.eu/web_Google.txt")
def traitementCcfWIthSecondarySorting(key, values):
    minVal = values[0]
    resultat= []
    if minVal < key:
        resultat.append((key,minVal))
        for value in values:
            if(minVal != value):
                resultat.append((value, minVal))
                acc.add(1)
    return resultat

entre=graphe.map(lambda x : x.split('\t'))
acc = sc.accumulator(1)
stop=1
while stop !=0:
    acc.value = 0
    grapheCcf = entre.flatMap(lambda x : [(x[0],x[1]),(x[1],x[0])]).groupByKey().map(lambda x: (x[0], sorted(x[1])))
    iterate = grapheCcf.flatMap(lambda x: traitementCcfWIthSecondarySorting(x[0], x[1]))
    dedup=iterate.map(lambda x:((x[0],x[1]),0)).reduceByKey(lambda x,y:x+y).flatMap(lambda x: [(x[0][0],x[0][1])] if x
    entre=dedup
    result = dedup.collect()
    stop=acc.value
print(dedup.count())
```

▸ (9) Spark Jobs

872967

Command took 8.07 minutes -- by mohamed-houssem.hamroun@dauphine.eu at 10/02/2021 à 22:37:43 on test

Le temps d'exécution du programme dans Data Bricks avec les données les plus massives est de 480 secondes. Le programme trouve 872967 composantes connexes.

PARTIE 6 : Résultats simplifiés :

Afin d'avoir une vue globale sur le développement pyspark réalisée, on va classifier nos méthodes par rapport à la quantité
de données en entrées, temps d'exécution, quantité de données en sortie, nombre de composantes connexes trouvées, environnement d'exécution (local, Data bricks, AWS Hadoop)

Algorithme 1 : Ccf iterate

Data	nombre de composante trouvées	temps d'exécution Hadoop AWS	temps d'exécution Data bricks	temps d'exécution en local	nombre de nœuds traitée
facebook.txt	4038	35	6	9	4,039
twitter.txt	81305	120	240	140	81,306
google.txt	872967	700	810	Non supportable	875,713

Algorithme 2 : Ccf iterate with sorting

Data	nombre de composantes connexes	temps d'exécution Hadoop AWS (en secondes)	temps d'exécution Data bricks (en secondes)	temps d'exécution en local (en secondes)	nombre de nœuds traitée
facebook.txt	4038	35	4	9	4,039
twitter.txt	81305	100	70	135	81,306
google.txt	872967	378	450	non supportable	875,713

L'utilisation maximum de mémoire n'est pas bénéfique pour Hadoop dans certain cas la configuration par défaut semble plus efficace. Néanmoins, selon les cas et les données, les configurations d'exécution changent.

partie 7 : implémentation du job MapReduce dans pyspark

Les deux fichiers mapper.py et reducer.py font une seule itération seulement. pour avoir le résultat final de l'algorithme j'ai créé un script Bash qui s'exécute au niveau de la machine virtuelle afin d'appeler les fichiers plusieurs fois tant que le compteur n'est pas mis à zéro.

Le script Bash est affiché ci-dessous.

```
#!/bin/bash
valid=1
fichier=$1
while [ $valid -eq 1 ]
do
cat $fichier |python mapper.py|python reducer.py>result.txt
tag=$( tail -n 1 result.txt)
if [ ${tag:0:1} == "0" ];
then
valid=0
fi

sed -i '$d' result.txt
cat result.txt>$fichier
cat $fichier
done
```

Ce script exécute les Job Map et Reduce dans la mémoire de la machine virtuelle. Après avoir importé les fichier mapper.py, reducer.py et le fichier script sur Hadoop il suffit de remplacer la partie d'exécution :

```
cat $fichier |python mapper.py|python reducer.py>result.txt
```

Par :

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input  /user/hadoop/wc/input \
-output  /user/hadoop/wc/output \
-file /home/hadoop/mapper.py \
-mapper /home/hadoop/mapper.py \
-file  /home/hadoop/reducer.py \
-reducer /home/hadoop/reducer.py
```

les logiciels utilisés :

Durant notre projet on a utilisé les logiciels suivantes
pycharm : pour développer et tester les différentes algorithmes
data bricks et Rosetta aws : pour tester les algorithmes sur des graphes d'une grande masse de données