

Rendu Projet Spark 21/09/2020

Algorithme du CCF en PySpark

Master IASD Dauphine-PSL 2020

Anthony Baltzinger
13/10/2020

Table des matières

I.	Introduction.....	2
1.	Contexte	2
2.	Rendu	2
3.	Exécution du script sur le cluster de Dauphine.....	2
4.	Version et variables d'environnement.....	2
II.	Implémentation.....	3
1.	Présentation de l'algorithme MAP REDUCE.....	3
a)	Présentation de l'algorithme	3
b)	CCF-Iterate et CCF-Dedup	3
2.	Implémentation Spark – RDD.....	5
3.	Implémentation Spark - Dataframe	9
III.	Résultats et conclusions.....	13
1.	Les dataset.....	13
2.	Les machines Spark	13
3.	Résultats	13
4.	Conclusion	14

I. Introduction

1. Contexte

Le but de ce projet est d'implémenter en Spark l'algorithme de recherche des composants connexes dans un graphe défini dans l'article de recherche [CCF: Fast and Scalable Connected Component Computation in MapReduce](#) paru en Février 2014.

L'objectif est de vérifier le passage à l'échelle de cet algorithme en augmentant la taille des graphes.

2. Rendu

Le rendu de ce TP est sous forme de dossier contenant :

- Le notebook utilisé sur Databricks
- Les scripts python contenant l'ensemble du code source
- Ce même compte-rendu

3. Exécution du script sur le cluster de Dauphine

En plus du notebook de Databricks, l'implémentation a été faite sur un script python afin de pouvoir le lancer sur un cluster spark tel que celui de dauphine.

Pour cela il suffit de se placer sur le master node spark et d'exécuter la commande :

```
spark-submit baltzinger_ccf_pyspark_rdd_1.py
```

4. Version et variables d'environnement

La version utilisée sur Databricks

- Spark 3.0.1
- Hadoop 3.3.0
- Python 3.7.5

Sur le cluster du LAMSADE, nous avons eu besoin de modifier les quelques variables d'environnement ci-dessous pour que l'interpréteur PySpark utilise la version 3 de python et non la 2.7 comme c'est le cas par défaut :

```
alias python=python3
export PYSPARK_PYTHON="/usr/bin/python3.7"
export PYSPARK_DRIVER_PYTHON="/usr/bin/python3.7"
```

II. Implémentation

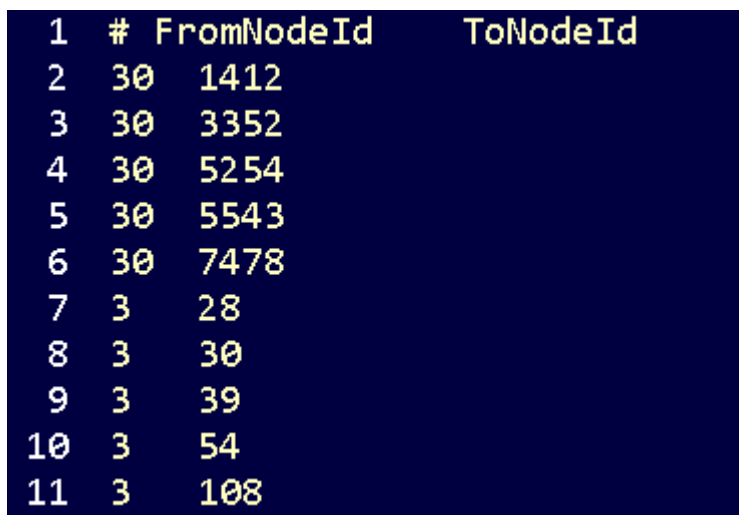
1. Présentation de l'algorithme MAP REDUCE

a) Présentation de l'algorithme

L'algorithme MapReduce est proposé dans l'article de recherche [CCF: Fast and Scalable Connected Component Computation in MapReduce](#).

Il prend en entrée un graphe non orienté, ce qui signifie que le nœud A est connecté à B est équivalent à dire que le nœud B est connecté à A. Pour les implémentations futures, ce graph sera enregistré sur un fichier plat (généralement au format texte) dont chaque ligne correspond à un lien entre deux nœud.

Voici un extrait d'un graphe sur ce format :



1	#	FromNodeId	ToNodeId
2	30	1412	
3	30	3352	
4	30	5254	
5	30	5543	
6	30	7478	
7	3	28	
8	3	30	
9	3	39	
10	3	54	
11	3	108	

Figure 1 - Extrait d'un graphe d'entrée

L'algorithme va parcourir par itération les nœuds du graphe en allant du plus grand nœud (i.e ; le nœud avec l'indice le plus fort) vers le plus petit nœud. Il crée à chaque itération de nouvelles paires de nœuds dans le graphe pour chaque nouvelle arête parcourue. Lorsque qu'aucune nouvelle paire a été créée, l'itération prend fin et le processus retourne le graphe avec l'ensemble des nœuds connexe du graphe.

b) CCF-Iterate et CCF-Dedup

Chaque itération s'appuie sur deux job MapReduce exécutés en séquence qui sont le CCF-Iterate et le CCF-Dedup.

Le job CCF-Iterate contient la majeure partie de l'algorithme :

- **Task MAP** : la task map renvoi l'ensemble des couples ainsi que les couples permutés, du fait que nous sommes sur un graphe non orienté
- **Task REDUCE** : la task reduce va regrouper l'ensemble des nœuds avec la liste des nœuds qui lui sont connexes et va ensuite rechercher le minimum entre ces deux

quantités. Si le nœud minimum se trouve dans les nœuds connexe, dans ce cas nous considérons ce nœud minimum comme étant le nouveau nœud de référence pour cette liste de nœud connexe. Dans ce cas le compteur de pair s'incrémente et une nouvelle itération aura lieu.

CCF-Iterate

```

map(key, value)
  emit(key, value)
  emit(value, key)

reduce(key, < iterable > values)
  min ← key
  for each (value ∈ values)
    if(value < min)
      min ← value
  valueList.add(value)
  if(min < key)
    emit(key, min)
    for each (value ∈ valueList)
      if(min ≠ value)
        Counter.NewPair.increment(1)
        emit(value, min)

```

Figure 2 - pseudo-code CCF-Iterate

Le job CCF-Dedup est, comme expliqué dans l'article, un rajout pour améliorer les performances de l'algorithme. En effet son but est de supprimer les doublons dans la liste du graphe introduit par le job CCF-Iterate.

CCF-Dedup

```

map(key, value)
  temp.entity1 ← key
  temp.entity2 ← value
  emit(temp, null)

reduce(key, < iterable > values)
  emit(key.entity1, key.entity2)

```

Figure 3 - pseudo-code CCF-Dedup

Les implémentations en Spark vont s'appuyer cet algorithme MapReduce et ces deux job CCF-Iterate et CCF-Dedup.

2. Implémentation Spark – RDD

a) 1^{ère} implémentation du CCF

Pour la suite de l'implémentation nous travaillerons sur le dataset Web-Google, il comporte 875 000 nœuds et 5,1 million d'arrêtes. Ce dataset est disponible sur le site de Stanford :

<http://snap.stanford.edu/data/web-Google.html>

Préparation des données

Le RDD de départ sera constitué du nœud de départ en key et d'un tableau des nœuds connexes en value :

```
1 # Suppression des premières lignes de description du dataset
2 # Conversion de ligne en un tableau clé-valeur
3 # Conversion des valeurs en tableau
4 graph = dataset.filter(lambda x: "#" not in x)\
5                     .map(lambda x : x.split("\t"))\
6                     .map(lambda x : (int(x[0]), [int(x[1])]))
7 graph.take(10)
```

► (1) Spark Jobs

```
Out[14]: [(0, [11342]),
(0, [824020]),
(0, [867923]),
(0, [891835]),
(11342, [0]),
(11342, [27469]),
(11342, [38716]),
(11342, [309564]),
(11342, [322178]),
(11342, [387543])]
```

Command took 0.77 seconds -- by anthony.baltzinger@dauphine.eu at 08/10/2020 à 16:51:03 on myCluster

CCF-Iterate MAP

```
# CCF-Iterate MAP
# On double Le RDD avec Les données permutées
ccf_iterate_map = graph.union(graph.map(lambda x : (x[1][0], [x[0]])))
```

CCF-Iterate REDUCE

```
# CCF-Iterate REDUCE
# Le reduce constitue Le tableau des noeuds connexes pour chaque noeud key
# Ensuite on détermine Le min entre Le noeud key et ses noeuds connexes
# Enfin on élimine Les lignes dont Le minimum correspond au noeud key (pas de
nouvelle paire potentielle)
ccf_iterate_reduce_pair = ccf_iterate_map\
    .reduceByKey(lambda x,y : x+y)\
    .map(lambda x : (x[0], x[1], min(x[0], min(x[1]))))\
    .filter(lambda x: x[0] != x[2])
```

Afin de déterminer si de nouvelles paires sont créées nous définissons une fonction qui compte les entrées restantes dans le RDD et qui mettra à jour un accumulateur :

```
newPair = sc.accumulator(0)

def countNewPair(x):
    global newPair
    start = 0
    for value in x[1]:
        if value != x[2]:
            start = start + 1
    newPair += start

# Calcul of new pairs created
ccf_iterate_reduce_pair.foreach(countNewPair)
```

Une fois ce calcul fait nous terminons le CCF-Iterate REDUCE en émettant les nouvelles paires constituées :

```
# CCF-Iterate REDUCE - fin
# Nous émettons les entrées restantes avec le noeud min en tant que noeud key
# et la liste des nœuds connexes
ccf_iterate_reduce = ccf_iterate_reduce_pair\
    .map(lambda x : (x[2], x[1] + [x[0]]))\
    .flatMapValues(lambda x : x)\
    .filter(lambda x : x[0] != x[1])\
    .map(lambda x : (x[0], [x[1]]))
```

CCF-Dedup MAP

```
# CCF-Dedup - MAP
# Nous émettons le couple key-value en key avec une valeur sample
ccf_dedup_map = ccf_iterate_reduce.map(lambda x : ((x[0], x[1][0]), None))
```

CCF-Dedup REDUCE

```
# CCF-Dedup - REDUCE
# Nous faisons un groupByKey pour éliminer les doublons
ccf_dedup_reduce = ccf_dedup_map\
    .groupByKey()\
    .map(lambda x : (x[0][0], [x[0][1]]))
```

Exécution sur Databricks

Sur le dataset web-Google, cette implémentation du CCF s'exécute en une 17 minutes pour 8 itérations :

```

▶ (8) Spark Jobs
Itération : 1 Number of newPair : 8552232
Itération : 2 Number of newPair : 13310683
Itération : 3 Number of newPair : 16589455
Itération : 4 Number of newPair : 20477909
Itération : 5 Number of newPair : 22383232
Itération : 6 Number of newPair : 22470015
Itération : 7 Number of newPair : 22471333
Itération : 8 Number of newPair : 22471333

Command took 17.12 minutes -- by anthony.baltzinger@

```

Figure 4 - Exécution sur Databricks - 1ère version RDD

Exécution sur le cluster du LAMSADE

Sur le cluster du LAMSADE, nous exécutons cette implémentation à l'aide du script **baltzinger_ccf_pyspark_rdd_1.py** et de la commande `spark-submit` pour une exécution en 7,3 minutes.

b) 2^e implémentation avec optimisation sur les RDD

Axe d'amélioration sur la 1^{ère} implémentation du code :

- Les nombreuses transformations sur le CCF-Iterate REDUCE
- L'utilisation de tableau de valeur pouvant grossir de manière inattendue sur certain graphe
- Les opérations du CCF-Dedup sont facilement réalisable en RDD avec l'opération **distinct()**

Afin d'améliorer ces performances, il est fait le choix de rester sur des entrées clé-valeur simple (tuple) pour les itérations du CCF, ainsi que de modifier la fonction **countNewPair** afin qu'elle réalise une partie des opérations de transformation sur le RDD.

Modification du RDD pour travailler sur un tuple simple pour chaque entrée

Cette modification implique de supprimer la transformation en tableau pour la partie valeur et de rester sur des tuples pour les itérations du CCF :


```

1 # Suppression des premières lignes de description du dataset
2 # Conversion de ligne en un tableau clé-valeur
3 # Conversion des valeurs en tableau
4 graph = dataset.filter(lambda x: "#" not in x)\
5     .map(lambda x : x.split("\t"))\
6     .map(lambda x : (int(x[0]), int(x[1])))
7 graph.take(10)

```

► (1) Spark Jobs

```

Out[7]: [(0, 11342),
(0, 824020),
(0, 867923),
(0, 891835),
(11342, 0),
(11342, 27469),
(11342, 38716),
(11342, 309564),

```

Simplification du CCF-Iterate REDUCE à l'aide de la fonction countNewPair

Notre fonction personnelle va toujours compter si de nouvelles paires ont été créées afin de déterminer si une nouvelle itération est nécessaire ou non. Cependant elle va également retourner l'ensemble des nouvelles paires générées à l'aide de la méthode **yield** :

```

# Nouvelle fonction countNewPair
def countNewPair(x):
    key = x[0]
    values = x[1]
    min = key
    valueList = []
    for value in values:
        if value < min:
            min = value
        valueList.append(value)
    if min < key:
        yield((key, min))
    for value in valueList:
        if min != value:
            accum.add(1)
            yield((value, min))

```

Cette nouvelle fonction **countNewPair** nous permet de simplifier la tâche du CCF-Iterate REDUCE comme suit :

```

# CCF-iterate REDUCE
ccf_iterate_reduce = ccf_iterate_map\
    .groupByKey()\
    .flatMap(lambda x: countNewPair(x))\
    .sortByKey()

```

Le **sortByKey()** sur cette dernière étape permet de simplifier l'opération de **distinct()** qui va suivre. En effet la nouvelle fonction **countNewPair** va générer un certain nombre de doublon et le fait d'ordonner le RDD groupe ces doublons et va faciliter leur élimination.

Simplification du CCF-Dedup

La partie du CCF-Dedup a pour but de supprimer les doublons dans le RDD après l'opération de recherche des nouvelles paires, or cette opération est implémentée de base dans Spark avec la fonction **distinct()**.

La partie CCF-Dedup peut donc se résumer à la ligne suivante :

```
# CCF-Dedup MAP & REDUCE
ccf_dedup_map_reduce = ccf_iterate_reduce.distinct()
```

Exécution sur Databricks

Ces simplifications ont permis d'améliorer l'exécution du code CCF avec l'analyse du dataset web-Google en 12 minutes, soit une réduction de 5 minutes par rapport à la 1^{ère} implémentation.

Exécution sur le cluster du LAMSADE

Sur le cluster du LAMSADE, nous exécutons cette implémentation à l'aide du script **baltzinger_ccf_pyspark_rdd_2.py** et de la commande spark-submit pour une exécution en 6,9 minutes.

3. Implémentation Spark - Dataframe

Pour cette partie nous reprendrons l'implémentation en utilisant les Dataframe de Spark ainsi que le langage SQL associé.


Passage du RDD au Dataframe

Nous reprenons le RDD d'origine que nous transformons en Dataframe :

```

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode, col, split, array, array_min, concat,
3
4 spark = SparkSession.builder.getOrCreate()
5
6 graph = dataset.filter(lambda x: "#" not in x)\
7                     .map(lambda x : x.split("\t"))\
8                     .map(lambda x : (int(x[0]), int(x[1])))
9
10 graph = spark.createDataFrame(graph).selectExpr("_1 as key", "_2 as value")
11 graph.printSchema()
12 graph.show()
13

```

►  df: pyspark.sql.dataframe.DataFrame = [key: long, value: long]

```

root
 |-- key: long (nullable = true)
 |-- value: long (nullable = true)

+----+-----+
|key|value|
+----+-----+
|  0|    2|
|  0|    3|
|  2|    0|
|  2|    6|
|  3|    0|
|  3|    6|

```

CCF-Iterate MAP

```

# CCF-Iterate MAP
# Union avec Le Dataframe où Les colonnes sont permutées
ccf_iterate_map = graph.union(\
    graph.select(col("value").alias("key"), col("key").alias("value")))

```

CCF-Iterate REDUCE

```

# CCF-Iterate REDUCE
# La fonction collect_set permet de regrouper Les valeurs dans un tableau
# tout en éliminant Les doublons de ce même tableau
# ensuite nous pouvons calculer Le min entre Le tableau et
# Le noeud key avec La fonction least
# puis nous filtrons si Le min est Le nœud key
ccf_iterate_reduce_pair = ccf_iterate_map\
    .groupBy(col("key")).agg(collect_set("value").alias("value"))\
    .withColumn("min", least(col("key"), array_min("value")))\
    .filter((col('key')!=col('min')))

```

Une fois les valeurs concaténées dans ce tableau, nous pouvons déterminer le nombre de nouvelles paires en calculant la taille du tableau obtenu car dépourvu de doublons :

```
# Calcul des nouvelles paires obtenues lors de l'itération
newPair += ccf_iterate_reduce_pair\
    .withColumn("count", size("value")-1)\
    .select(sum("count")).collect()[0][0]
```

Une fois ce calcul fait nous terminons le CCF-Iterate REDUCE en émettant les nouvelles paires constituées et en formant le Dataframe avec les colonnes key-value :

```
# CCF-Iterate REDUCE - fin
# Nous concaténons Le noeud key avec la liste des noeud connexe avec concat
# Nous gardons donc Le noeud minimum constituant notre nouveau noeud key
# Nous générons Les paires entre le noeud min et ses noeuds connexe avec explode
ccf_iterate_reduce = ccf_iterate_reduce_pair\
    .select(col("min").alias("a_min"),\
        concat(array(col("key")), col("value")).alias("valueList"))\
    .withColumn("valueList", explode("valueList"))\
    .filter((col('a_min')!=col('valueList')))\
    .select(col('a_min').alias("key"), col('valueList').alias("value"))
```

CCF-Dedup MAP & REDUCE

```
# CCF-Dedup MAP & REDUCE
# Utilisation du distinct pour supprimer Les doublons
ccf_dedup_reduce = ccf_iterate_reduce.distinct()
```






Exécution sur Databricks

Cette implémentation s'exécute en 26 minutes environ sur Databricks, nous avons donc une perte de performance par rapport à l'implémentation avec les RDD.

Exécution sur le cluster du LAMSADE

Sur le cluster du LAMSADE, nous exécutons cette implémentation à l'aide du script **baltzinger_ccf_pyspark_dataframe.py** et de la commande spark-submit pour une exécution en 14,8 minutes.

► (8) Spark Jobs

-  ccf_iterate_map: pyspark.sql.dataframe.DataFrame = [key: long, value: long]
-  ccf_iterate_reduce_pair: pyspark.sql.dataframe.DataFrame = [key: long, value: array ... 1 more fields]
-  ccf_iterate_reduce: pyspark.sql.dataframe.DataFrame = [key: long, value: long]
-  ccf_dedup_reduce: pyspark.sql.dataframe.DataFrame = [key: long, value: long]
-  graph: pyspark.sql.dataframe.DataFrame = [key: long, value: long]

```
Itération : 1 Number of newPair : 7223780
Itération : 2 Number of newPair : 11982231
Itération : 3 Number of newPair : 15261003
Itération : 4 Number of newPair : 19149457
Itération : 5 Number of newPair : 21054780
Itération : 6 Number of newPair : 21141563
Itération : 7 Number of newPair : 21142881
Itération : 8 Number of newPair : 21142881
```

Command took 26.47 minutes -- by anthony.baltzinger@dauphine.eu at 11/10/2020 à 22:24:01 on

III. Résultats et conclusions

1. Les dataset

Pour comparer les différences de temps d'exécution entre les différentes implémentations, nous utiliserons les dataset suivants, tiré du site de Stanford et trié par taille croissante :

Dataset	Nœud	Arrête	Diamètre	Lien
wiki-Vote	7115	103689	7	wiki-Vote
web-NotreDame	325 729	1,5 M	46	web-NotreDame
web-Google	875 000	5,1 M	21	web-Google
soc-Pokec	1,6 M	30 M	11	soc-Pokec

2. Les machines Spark

Pour comparer les exécutions, nous changerons également les environnements Spark utilisé en travaillant sur une ressource Databricks (Community), une installation Spark en local, et sur le cluster du LAMSADE à Dauphine :

Machine	Version	capacité	nb executor
Databricks	spark.3.0.1.hadoop.3.3.0	15.3 GB Memory, 2 Cores	1
PC local	spark-3.0.1-bin-hadoop3.2	6.9 GB Memory, 8 Cores	1
Cluster Dauphine	spark-3.0.0-preview2-bin-hadoop2.7	64 GB Memory	1

3. Résultats

Voici les temps d'exécution du CCF en fonction des implémentations :

Dataset	Implémentation	Databricks	PC local	Cluster LAMSADE
wiki-Vote	RDD_V1	19,87	11,92	14,68
	RDD_V2	32,14	40,53	33,45
	Dataframe	49,09	1350,55	125,17
web-NotreDame	RDD_V1	247,2	122,49	117,45
	RDD_V2	238,8	209,30	150,86
	Dataframe	397,2	> 1h	462,13
web-Google	RDD_V1	1027,2	459,02	438,53
	RDD_V2	811,2	854,39	415,38
	Dataframe	1588,2	> 1h	889,24
soc-Pokec	RDD_V1	> 1h	2285,75	> 1h
	RDD_V2	> 1h	1277,01	2035,19
	Dataframe	> 1h	> 1h	> 1h

4. Conclusion

Ce projet a permis de mettre en pratique l'algorithme du CCF sur spark avec l'utilisation des RDD et des Dataframe.

Les temps d'exécution sont proches des résultats de l'article qui parvient à un traitement du dataset web-Google en 256 secondes contre environ 400 secondes pour notre implémentation sur le cluster du LAMSADE.

Nous notons qu'entre les deux implémentations sur les RDD, nous avons des performances similaires voire moindre avec la V2 sur des graphes de plus petite taille, mais le gain se présente sur les graphes de plus grande taille. Ceci confirme que la 2^e implémentation a une meilleure scalabilité sur la taille de nos données. Il est d'ailleurs la seule implémentation à avoir une durée d'exécution inférieure à 1h pour le plus grand dataset.

De la même manière, les Dataframe sont sensé être plus performant que les RDD sur le traitement des données. Bien que notre implémentation s'exécute en un temps raisonnable même pour de grand dataset, nous remarquons que l'implémentation des RDD a de meilleures performances peu importe l'environnement ou le dataset.

Aussi nous remarquons qu'au vue de la taille restreinte des dataset nous n'utilisons pas le plein potentiel de Spark et du traitement distribué sur nos implémentations car nous obtenons de bonnes performances sur une machine local en comparaison avec le cluster du LAMSADE.

En effet l'exécution ne dépasse pas les 8 Gb de la machine local et permet la bonne exécution de notre implémentation. Un dataset plus volumineux aurait nécessité une gestion plus sérieuse de la mémoire avec l'utilisation de `persist()` et une surveillance des partitions générée par Spark durant les itérations.

Les temps d'exécution sont toutefois indicatifs car les ressources ne sont pas toujours « fiable », et des différences de temps d'exécution ont pu être observées à certain moment du projet. En effet le cluster Databricks étant en version gratuite nous n'avons pas d'assurance de la stabilité des ressources qui sont partagées.

Ce projet nous a montré qu'il est toutefois possible de rapidement mettre en application l'algorithme du CCF à l'aide de PySpark, et d'analyser des graphes d'une certaine taille, comme le web-Google, sur un cluster Community (gratuit) de Databricks. L'accessibilité des ressources et la simplicité de manipulation de Python et Spark simplifie grandement une première mise en œuvre ou des travaux d'exploration d'algorithme traitant des données massives.