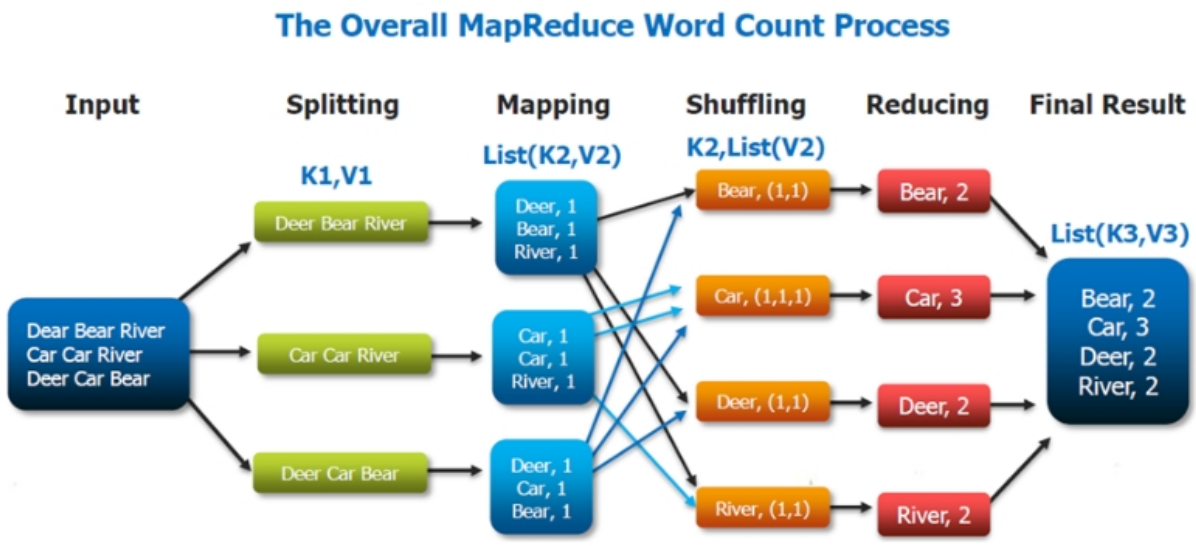
Rapport de projet ELTDM – Sujet : PageRank

Introduction

L'objet de ce document est de présenter une implémentation de l'algorithme Page Rank, principal algorithme à l'origine du moteur de recherche de Google, développé par Larry Page, son co-fondateur. Dans ce document, je vais présenter la manière dont j'ai codé l'algorithme, selon plusieurs approches (en m'inspirant parfois d'exemples en ligne), en tentant de le faire dans le cadre du paradigme Map Reduce de parallélisation des calculs. Tout d'abord, il convient de rappeler en quoi consiste Map Reduce, à travers un simple schéma représentant le « Hello World » de cette approche, à savoir word count.

Présentation de Map Reduce à travers l'exemple de word count



Map Reduce est un algorithme de calcul distribué développé de manière à répartir des traitements de données massives sur un ensemble de machines (appelé Cluster), selon une logique « clé-valeur ». Cette algorithme contient différentes étapes :

1. Split lines : l'algorithme lis un fichier ligne par ligne et stocke chaque ligne sur un nœud du cluster : la clé peut être l'identifiant de la ligne et la valeur la ligne sous forme de chaîne de caractères.

2. L'étape de Mapping consiste à séparer chaque mot de la chaîne de caractère (avec par exemple comme séparateur un espace – parfois on exclut les mots inférieurs à une certaines taille minimale) et à créer des couples clé-valeur de type (mot,1), et ce, pour tout mot de la ligne.

3. Shuffle and sort consiste à regrouper chaque mot avec une liste de « 1 » contenant autant de « 1 » que d'occurrence du mot

4. Reduce consite à agréger les valeurs du mot clé à travers une opération d'agrégation, dans le cas de word count il s'agit d'une somme des valeurs, c'est à dire la somme des 1 (représentant ainsi le nombre d'occurrence du mot)

5. La liste finale contient ainsi des couples (mot, nombre d'occurrence), répondant à l'objectif de word count.

Dans la suite de ce rapport, je vais détailler l'approche Map Reduce que je vais tenter de développer avec Python, en particulier avec la librairie PySpark.

Page Rank : cadre théorique

L'algorithme Page Rank a pour objectif de classer les pages web selon leur importance. Leur importance est quantifiée par le nombre de liens pointant vers ces dernières. Toutefois, afin d'éviter que des parasites ne créent des pages web fantômes pointant vers un site pour le faire monter artificiellement, l'algorithme prend en compte l'importance des pages pointant vers la page en question. Ainsi, plus un site est mentionné par un site ayant un rang élevé, plus il aura un rang important. Enfin, afin de prendre en compte le comportement des utilisateurs depuis une page web vers une autre, un « random surfer » aura une probabilité de visiter un lien égale à l'inverse du nombre de liens sur la page où il se situe.

Ainsi, la formule du page rank peut se formuler mathématiquement de la façon suivante :

$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

où |G| est le nombre de nœuds du graphe, α est le facteur de « random jump » ou téléportation, L(n) est l'ensemble des pages reliées à n et C(m) est le nombre de liens sur m (ie le nombre de liens sortants). La probabilité d'arriver sur n en partant de m est P(m)/C(m). On fait la somme pour obtenir le rang (en tenant compte du facteur de téléportation et du nombre de nœuds total.

Page Rank en Map Reduce

La formule du Page Rank suivant l'algorithme Map Reduce peut être formulée de la façon suivante :

1. Un Mapper reçoit un couple clé-valeur composé d'un nœud et de l'ensemble des liens sortants du nœud considéré. Le Mapper va procéder à 2 émissions : d'une part, le Page Rank initial avec son rang initial (ie l'inverse du nombre de nœuds) et, d'autre part, le couple contenant le nœud et la liste des liens sortants de ce dernier.

2. La partie Shuffle and sort consiste en un regroupement des page ranks intermédiaires de chaque noeud

3. Le reducer va ensuite recevoir ce couple et calculer le Page Rank en faisant la somme des contributions susmentionnées et va émettre les couples (nœud, PageRank) ainsi que le coupke (nœud, liens sortants) de nouveau.

Le pseudo-code ci-dessous résume les étapes précédentes sous forme algorithmique.

Algorithm 5.3 PageRank (simplified)

In the map phase we evenly divide up each node's PageRank mass and pass each piece along outgoing edges to neighbors. In the reduce phase PageRank contributions are summed up at each destination node. Each MapReduce job corresponds to one iteration of the algorithm. This algorithm does not handle dangling nodes and the random jump factor.

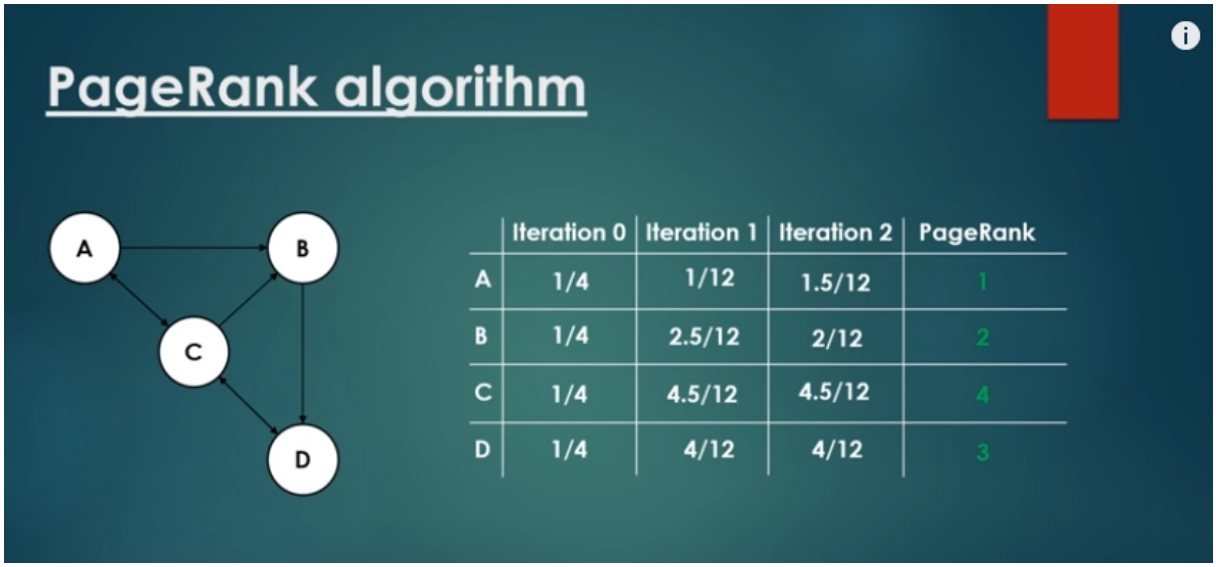
```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Les implémentations

Le précédent algorithme a été développé en suivant cette approche aussi rigoureusement que possible. J'ai mis au point un premier algorithme qui suit la logique Map Reduce en termes d'étapes pour bien comprendre les structures que le code requiert, et ce, afin de bien tout recoder (sans nécessairement toutefois faire appel à des fonctions parallélisant les calculs par méconnaissance de l'ensemble de ces fonctions initialement). Dans un deuxième temps, après avoir testé et compris le fonctionnement de certaines fonctions et structures propres à Spark, telles que GroupByKey, ReduceByKey ou encore les objets de type « pyspark.resultIterable » , je présente le fonctionnement d'une 2ème approche qui converge de la même manière, en la recodant en puisant des ressources externes que j'analyse (en décomposant chaque étape de manière à décrypter leur fonctionnement intégral), de manière à expliquer une approche plus optimisée et parfaitement parallélisée.

Afin de vérifier mes résultats, je m'appuie sur le graphe ainsi que les scores ci-dessous pour chaque itération, de manière à contrôler mes résultats à chaque étape. Dans le premier cas je ferai les 2 itérations et dans le second une seule pour contrôler la convergence des résultats vers les mêmes valeurs.



Dans le cas présenté ci-dessus, le random surfer n'est pas implémenté. Toutefois, il suffit de multiplier ce résultat par d (ex : d= 0.85) et d'ajouter (1 – d) (ie 0.15) pour obtenir une formule correspondant à celle présentée initialement (à ceci près que le random surfer est divisé par N dans la formule théorique, mais plusieurs implémentations ne le prennent pas nécessairement en compte).

Pour chaque étape, j'imprime sur la sortie standard mes opérations pour contrôler ce que je fais. L'affichage peut paraître lourd mais il permet de visualiser les étapes, y compris pour corriger c'est plus pratique.

Première approche du Page Rank

Etape 1 : Approche Mapping

Le code est contenu dans le fichier pageRank1.py et détaille chaque opération, de manière à comprendre la logique. Le code est structuré en X étapes. La première consiste à créer les structures initiales nécessaires à l'étape de mapping. Après exécution du code, une pause de 10 secondes (time.sleep(10)) permet de visualiser ces paramètres. Voici ce que l'on peut voir :

```
**************************************************************************************************************
---------------------------------------- FIN ETAPE 1 : RAPPEL des parametres initiaux  ----------------------
---------

---- liste des pages distinctes couplee a la liste des pages qui pointent vers elles ----
[[u'B', [[u'A', 0.0], [u'C', 0.0]]], [u'C', [[u'A', 0.0], [u'D', 0.0]]], [u'D', [[u'B', 0.0], [u'C', 0.0]]], [u
<type 'list'>
---- RANKS at time = 0 ----
<type 'list'>
[[u'A', 0.25], [u'C', 0.25], [u'B', 0.25], [u'D', 0.25]]
4
---- NB LIENS sortants de chaque page---
[[u'A', 2], [u'B', 1], [u'D', 1], [u'C', 3]]

**************************************************************************************************************
```

On y trouve :

- Une structure qui contient le couple (url, listes des pages pointant vers cette url)
- Les rangs initiaux
- Le nombre de liens sortants par page

Etape 2 : Approche Reducing

Dans cette étape, on effectue 2 itérations au cours desquelles on va parcourir :

- La liste des pages couplées à la liste de celles qui pointent vers elle
- La liste des couples (url, nombre de liens sortants)

Tout en combinant ces informations pour calculer le Page Rank à chaque étape.

A l'issue de l'itération 1, on obtient :

```
ranks-list-temp
[[u'A', 0.08333333333333333], [u'C', 0.375], [u'B', 0.20833333333333331], [u'D', 0.333333333333333
```

Ce qui correspond bien au résultat annoncé précédemment, avec, par ailleurs, la somme des rangs qui vaut 1.

A l'issue de l'itération 2, on obtient :

```
[[u'A', 0.125], [u'C', 0.375], [u'B', 0.16666666666666666], [u'D', 0.3333333333333333]]
1.0
ranks_init avec *0.85 + 0.15 :
[[u'A', 0.25625], [u'C', 0.46875], [u'B', 0.29166666666666663], [u'D', 0.43333333333333335]]
1.45
-------- termine --------------
[surena@lyn1e338-hdp pageRank]$
```

Ce qui correspond bien au résultat annoncé précédemment, avec, par ailleurs, la somme des rangs qui vaut 1. De plus, On calcule les Page Ranks avec la random surfer juste en dessous.

Conclusion : on obtient bien les bonnes valeurs et on peut observer les différentes étapes du code avec des « print » à chaque étape. Toutefois, le code n'est pas optimal car il ne profite pas suffisamment de la parallélisation de calculs offerts par Spark. De plus, certaines structures ne correspondent pas exactement aux structures prévues théoriquement.

Ainsi, il convient d'étudier à une meilleure approche en termes de parallélisation. Il s'agit de l'objet de l'étape 2.

Seconde approche du Page Rank

Afin de surmonter les premières limites, il convient d'introduire de nouvelles fonctionnalités offertes par Spark en matière de parallélisation de calculs. Le code se trouve dans PageRank2Final.py
Ce code contient les mêmes étapes que le précédent jusqu'à la 3ème (rien ne change vraiment en termes de méthode et surtout de logique), à ceci près qu'il contient une optimisation à l'étape 4 au niveau du Map Reduce (par rapport au code précédent), en ce sens qu'il profite pleinement de la puissance de parallélisation de Spark à travers certaines spécificités que j'analyse dans la suite du document.

Etape 1: rien ne change
En effet, il s'agit à l'origine de préparer les données en les lisant tout en récupérant les lignes, avec la méthode textFile du SparkContext.

Etape 2: rien ne change de nouveau
Ensuite, on crée les rangs initiaux en isolant les urls avec un premier groupByKey

Etape 3 : idem, rien de neuf
Puis, en employant, un MAP récupérant la lettre (en position 0 au sein de la structure résultant du groupByKey) tout en associant cette dernière à l'inverse du nombre total de nœuds au sein d'une structure de type « tuple » (ce qui créé un couple de type (url, 0.25) avec 4 nœuds distincts).

Etape 4 : la nouveauté
Ce qui diffère ici est l'optimisation permise par un objet nouveau type : le ResultIterable (cf https://spark.apache.org/docs/1.1.0/api/python/pyspark.resultiterable.ResultIterable-class.html).

J'ai découvert ce résultat dans le cadre de ce tutoriel : https://www.coursera.org/lecture/big-data-analysis/pagerank-optimization-gGbZo

J'ai tenté de décomposer chaque étape générée par l'opération contenue dans la variable ''scores''.

Tout d'abord, links.join(range) procède à une jointure entre links et rangs_t0 (contenant les rangs initiaux). Que se passe-t-il ? Le code ci-dessous permets de détailler le contenu de cette structure complexe et opaque:

```
analyze = links.join(rangs_t0).collect()
for a in analyze:
        for b in a:
                if isinstance(b,tuple):
                        for c in b:
                                if isinstance(c,float):
                                        print(c)
                                else:
                                        for k in c:
                                                print("result iterable contains :")
                                                print(k)
                else:
                        print(b)
```

Il en ressort ceci :

```
A
result iterable contains :
B
result iterable contains :
C
0.25
D
result iterable contains :
C
0.25
C
result iterable contains :
D
result iterable contains :
A
result iterable contains :
B
0.25
B
result iterable contains :
D
0.25
```

Pour la première lettre, on voit qu'on a un A en position 0 puis, en position 1, un tuple contenant B et C qui sont les nœuds vers lesquels pointe A ainsi que le rang de A. De même, C est de rang 0.25 et pointe vers D, A et B.

Qu'en déduit-on ? Cette structure permet de stocker pour chaque lien son rang ainsi que l'ensemble des liens vers lesquels il pointe.

A partir de cet élément, on peut appliquer une fonction de **Mapping** couplée à une fonction récupérant l'élément 1 de chaque composant de links.join(rangs_t0), c'est à dire la liste des liens sortant de l'url0 située quant à elle en position 0 de la structure gloabale ainsi que le score de cette dernière, de manière à assigner à chaque url de la sous liste d'urls située en position 0 du couple situé en position 1 de la structure globale le rang initial (ici 0.25) divisé par le nombre de liens sortant de l'url0. L'Output correspond ainsi à un ensemble de couples (tuples) contenant chacun une url de la liste urls ainsi qu'un Page Rank intermédiaire, en l'occurrence le Page Rank du lien qui pointait vers celle située en position 0 du nouveau couple en question, divisé par le nombre de liens sortant de celle-ci. Plus concrètement, on avait une structure avec une url0, un couple contenant une liste d'urls notée urls équivalent aux liens sortants de url0 et le rang de url0. Pour chaque url de la liste urls, url0 pointe vers chaque url : or pour le calcul du Page Rank de ces url, on doit prendre les liens qui pointent vers chacune d'elles, en les pondérant par le nombre de leurs liens sortant. C'est précisément ce que l'on fait. Les liens url0 desquels sortent les url de la liste urls deviennent ici les liens qui pointent vers les url de la liste urls. Ainsi, on obtient un Page Rank pour chaque url à partir du Page Rank de l'url0 qui pointe vers elle, pondéré par le nombre de liens sortants de url0 (ie la taille de la liste urls).

La commande suivante réalise cette opération de Mapping :

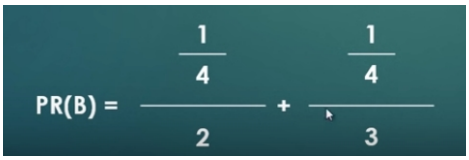
```
scores = links.join(rangs_t0).flatMap(lambda x: getScore(x[1][0], x[1][1]))
```

On obtient :

```
[(u'B', 0.125), (u'C', 0.125), (u'C', 0.25), (u'D', 0.08333333333333333), (u'A', 0.08333333333333333), (u'B', 0.08333333333333333), (u'D', 0.25)]
```

Ainsi, en faisant cette opération pour toutes les sous url de la liste globale, on calcule l'ensemble des rangs des url pondérées par leur nombre de liens sortants. Il suffit de les regrouper par clé (**REDUCE**), ie par url tout en additionnant chacun de leur éléments en position 1 dans le couple qu'ils forment (ie

le rang)(shuffle) et de les sommer (reduce via « add ») pour obtenir le rang de chaque url (comme somme des rangs des urls pointant vers chacune d'elles et pondérés par le nombre de liens sortants, comme dans l'image ci-dessous : somme des 2 urls pointant vers B, pondérées par le nombre de lien sortant de chacune d'elles).



Cette opération d'addition des rangs intermédiaires par clé est réalisée par l'opération reduce suivante :

```
new_ranks = scores.reduceByKey(add)
```

Le résultat obtenu est le suivant :

```
[(u'A', 0.08333333333333333), (u'D', 0.3333333333333333), (u'C', 0.375), (u'B', 0.20833333333333331)]
1.0
```

On obtient la même chose que dans la première approche pour la première itération. La somme des rangs vaut également (pour prendre en compte le random surfer, il suffit de multiplier ce score par 0.85 et de lui ajouter 0.15).

CONCLUSION :

La compréhension des opérations de jointures et des objets de type « resultIterable » de Pyspark permet de réaliser, à l'aide des opérations de mapping et de reduceByKey/GroupByKey, un PageRank tenant sur 2 lignes et parfaitement optimisé au paradigme MapReduce. La réalisation d'un PageRank initial, indépendamment de ces opérations, m'a permis de comprendre la structure et le fonctionnement interne des briques de PageRank, tout en comprenant (par une analyse approfondie et de la rétro-ingénierie) et implémentant une manière plus optimisée de coder l'algorithme Page Rank (adaptée à PySpark et aux clusters HDFS). La convergence vers les mêmes valeurs de ces 2 approches empiriques, en comparaison des résultats obtenus théoriquement, confirme le bon fonctionnement de ces algorithmes.