

# Large Scale Data Managment

Thibault Béziers la Fosse, Dennis Bordet

Decembre 2016

## 1 Introduction

Dans ce projet nous avons eu pour objectif d'implémenter l'algorithme *PageRank*, et de l'exécuter sur Hadoop et Spark, sur des données.

Les données récupérées étant relativement indigestes, nous avons dû les parcourir avec un script *Python* afin de pouvoir les exploiter.

Les données utilisées sont issues du Common Crawl, cependant le script permettant de les nettoyer provient d'un autre binôme. Effectivement, leur script étant efficace, nous avons jugé bon de le récupérer, et de le modifier afin qu'il effectue ce que nous désirions, pour obtenir les données de notre choix.

Notons aussi que dans l'algorithme mis en place, nous retirons les sites se référant eux-mêmes.

Dans ce rapport nous parlerons de l'algorithme que nous avons mis en place, suivi de nos résultats d'exécution, et enfin une conclusion sur notre projet.

## 2 Algorithme

Dans cette partie nous détaillerons l'algorithme que nous avons implémenté en *PIG*. Nous verrons, étape par étape, son exécution sur des données (réduites).

Voici les données que nous allons parcourir, elles sont sous la forme site - pagerank - références:

```
http://A.com/ 1 {(http://B.com/),(http://C.com/),(http://D.com/),(http://E.com/)}
http://B.com/ 1 {(http://C.com/)}
http://C.com/ 1 {(http://D.com/),(http://A.com/),(http://E.com/)}
http://D.com/ 1 {(http://C.com/)}
http://E.com/ 1 {(http://C.com/)}
```

Correspondant au système suivant:

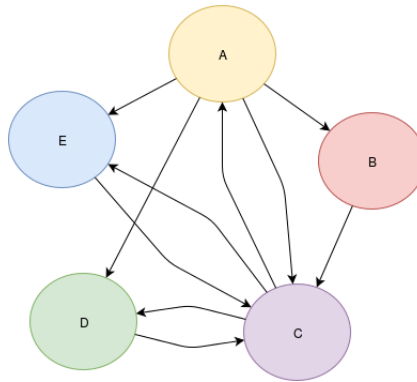


Figure 1: Liaisons entre les sites

On commence ainsi par les lire, afin de créer une table correspondante:

```

initial =
  LOAD 'data/file.txt'
  USING PigStorage('\t')
  AS ( url: chararray, pagerank: float, links:{ link:tuple(url: chararray) } );

```

De cette manière, *Pig* génère trois colonnes, en lisant le fichier, les lignes séparées par des tabulations:

| url           | pagerank | links   |
|---------------|----------|---|
| http://A.com/ | 1.0      | (http://B.com/), (http://C.com/),<br>(http://D.com/), (http://E.com/) |
| http://B.com/ | 1.0      | (http://C.com/)   |
| http://C.com/ | 1.0      | (http://D.com/),(http://A.com/),(http://E.com/)                       |
| http://D.com/ | 1.0      | (http://C.com/)   |
| http://E.com/ | 1.0      | (http://C.com/)   |

On exécute ensuite :

```

flattened_pgrnk =
  FOREACH initial
  GENERATE
    pagerank / COUNT ( links ) AS pagerank,
    FLATTEN ( links ) AS to_url;

```

Ensuite, on génère autant de tuples que de liens par url, et on lie chaque lien au pagerank de l'url qu'il indexe, divisé par le nombre de lien que cette url indexe.

On obtient ainsi le résultat suivant:

| pagerank   | link          |
|------------|---------------|
| 0.25       | http://B.com/ |
| 0.25       | http://C.com/ |
| 0.25       | http://D.com/ |
| 0.25       | http://E.com/ |
| 1.0        | http://C.com/ |
| 0.33333334 | http://D.com/ |
| 0.33333334 | http://A.com/ |
| 0.33333334 | http://E.com/ |
| 1.0        | http://C.com/ |
| 1.0        | http://C.com/ |

Enfin on exécute:

```
grouped_pgrnk =
FOREACH ( COGROUP flattened_pgrnk BY to_url, initial BY url INNER )
GENERATE
group AS url,
    0.2+0.5*SUM(flattened_pgrnk.pagerank) AS pagerank,
    FLATTEN ( initial.links ) AS links;
```

Enfin on groupe le tableau précédent par lien, et pour chaque url, on fait la somme des pageranks obtenus, divisés par deux, à laquelle on ajoute 0.2.

On obtient ainsi le tableau avec les pageranks par page, après une seule itération:

| url           | pagerank           | links   |
|---------------|--------------------|---|
| http://C.com/ | 2.475              | (http://D.com/),(http://A.com/),(http://E.com/)                     |
| http://D.com/ | 1.1416666716337205 | (http://C.com/)   |
| http://E.com/ | 1.1416666716337205 | (http://C.com/)   |
| http://A.com/ | 1.0166666716337205 | (http://B.com/),(http://C.com/),<br>(http://D.com/),(http://E.com/) |
| http://B.com/ | 0.975              | (http://C.com/)   |

Ce tableau ayant la même structure que le premier, on peut ainsi effectuer de nouvelles itérations dessus, jusqu'à ce que le pagerank tende suffisamment vers une valeur  $\lambda$ .

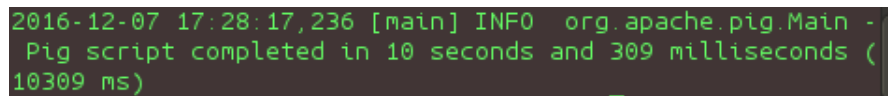
Notons que les résultats peuvent changer selon les valeurs que l'on donne au *damping factor* (0.5 ici). De plus on pourrait prendre en compte certains paramètres externes (sites favoris, browser...) .

### 3 Résultats

Dans cette section, nous comparerons l'exécution du même algorithme pagerank, sur Hadoop et Spark.

Nous utilisons des données issues du *Common Crawl*, avec environ 10 000 tuples, et sur une seule machine du CIE.

Voici ci-dessous des captures d'écran de nos résultats d'exécution:



```
2016-12-07 17:28:17,236 [main] INFO org.apache.pig.Main -  
Pig script completed in 10 seconds and 309 milliseconds (10309 ms)
```

Figure 2: Exécution avec Pig



```
720ms  
scala> █
```

Figure 3: Exécution avec Spark

On remarque que les temps d'exécutions sont très différents. Effectivement, Spark est dix fois plus rapide sur cette exécution.

### 4 Conclusion

Nous tenons à remercier Arnaud Graal, Lenny Lucas et Thomas Minier pour leur aide lors de l'implémentation des algorithmes de *PageRank*.

En conclusion, l'implémentation de l'algorithme Pagerank a été très instructive. Effectivement, malgré notre utilisation quotidienne des moteurs de recherche, nous ne connaissons pas l'algorithme répertoriant nos sites web favoris.

Nous avons pu voir les différences entre Spark et Hadoop, et bien que nous sachions les différences, d'un point de vue théorique, voir un temps d'exécution dix fois plus petit chez Spark a été assez surprenant, presque amusant. Et encore, ces exécutions étaient en local, nous imaginons une différence tout aussi impressionnante à grande échelle.

Enfin, nous pensons donc qu'entre Hadoop et Spark, il vaut mieux choisir ce dernier, tant pour sa simplicité d'utilisation que ses performances.

## 5 Ressources

**Github:** <https://github.com/orichalque/pagerank>