Big Data, Hadoop, MapReduce - Introduction pour statisticien non-initié

Jérémie Bigot et Adrien Richou

02/12/2016

Préambule

Ce document est une introduction à la problématique dite du Big Data, et de la gestion et extraction d'informations de données distribuées à partir des principes du logiciel Hadoop et de la programmation MapReduce. Afin d'illustrer ces notions, l'ensemble du document s'appuie sur le langage R et l'utilisation de la librairie `rmr2` de RHadoop. Dès à présent, nous avertissons le lecteur du caractère non-ambitieux de ce document dans la mesure où l'ensemble des traitements de données à partir de MapReduce et Hadoop sont réalisés en local sur une seule machine. Le passage à l'échelle des exemples traités dans ce document sur un serveur Hadoop distant est (pour le moment) bien au délà des compétences des auteurs.

Une part importante de ce document s'inspire d'articles et tuteuriels réalisés par des enseignants-chercheurs de l'Université de Toulouse :

- Besse P.(2016). Apprentissage de données massives, CIRM, Statistical Learning, Conférence invitée.
- Besse P., Laurent B.(2015). De Statisticien à Data Scientist; développements pédagogiques à l'INSA de Toulouse, Statistique et Enseignement, Vol. 7(1), 75-93.
- Besse P., Villa-Vilaneix N.(2014). Statistique et Big Data Analytics; Volumétrie, L'Attaque des Clones.
- Site WikiStat : http://www.math.univ-toulouse.fr/~besse/Wikistat/

Ce cours s'appuie sur un retour d'expérience dans l'encadrement d'une série de TP en RHadoop donnés à des étudiants de Master 2 en Modélisation Statistique et Stochastique à l'Université de Bordeaux. Les deux contributions principales de ce document sont de détailler (sur quelques exemples simples) la façon dont des données (transformées en fichier `HDFS`) sont structurés en blocs par la librairie `rmr2`, et d'illustrer les problématiques que peuvent poser la programmation MapReduce pour des étudiants en statistique à partir de l'analyse de jeux de données réelles.

Contents

1 Qu'est-ce que le Big Data ?

Sur Wikipédia, on peut trouver la definition suivante.

Definition 1 (Wikipedia) *"Le big data, littéralement grosses données, ou mégadonnées, parfois appelées données massives, désignent des ensembles de données qui deviennent tellement volumineux qu'ils en deviennent difficiles à travailler avec des outils classiques de gestion de base de données ou de gestion de l'information."*

Fort d'une telle définition, le big data se caractérise alors par "les trois V" : Volume, Variété, Vélocité que l'on peut tenter de caractériser comme suit.

Volume - Tout d'abord il est important de préciser la notion de volume d'un ensemble de données. Il ne s'agit pas juste d'un fichier "trop gros pour être ouvert sur un tableur (Excel)" qui est un problème de stockage en mémoire vive. Les volumes en jeu dans le Big Data sont liés à une explosion

des données numériques créées dans le monde lors des dernières années : $1,2 * 10^{21}$ octets en 2010, $1,8 * 10^{21}$ octets en 2012, prévision de $40 * 10^{21}$ octets en 2020.

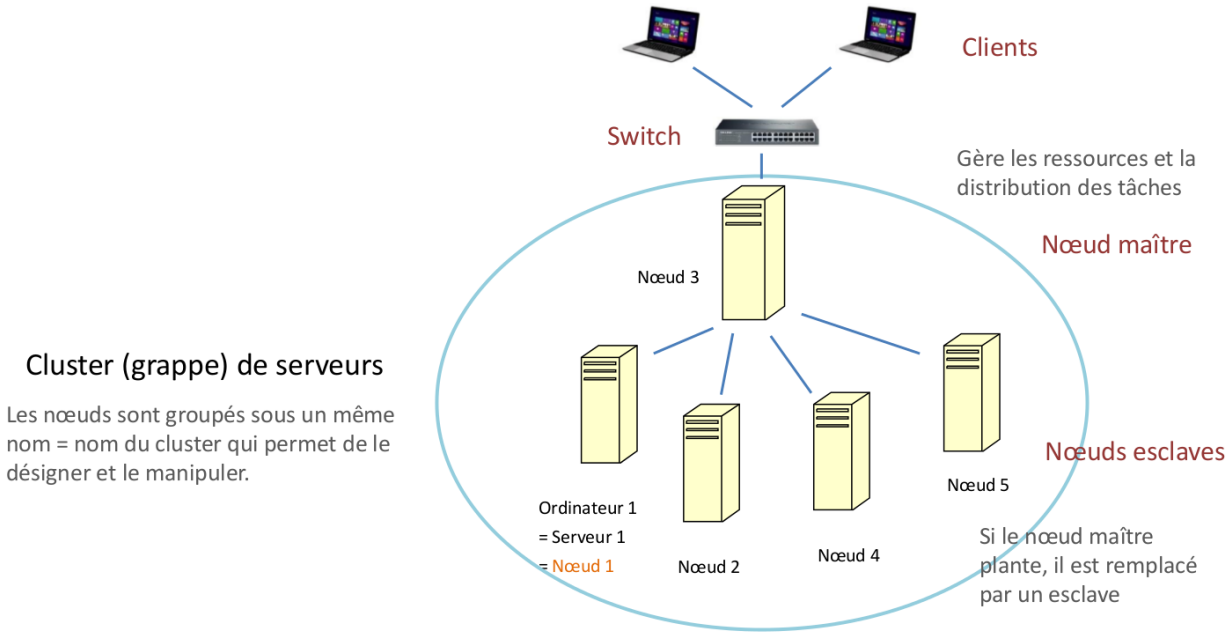
Ceci est lié aux grand acteurs d'Internet et également à des moyens d'enregistrement de données de plus en plus importants. En 2013, Twitter génère 7 téraoctets de données par jour, Facebook 10 téraoctets. Le Radiotélescope "Square Kilometre Array" (Australie) prévu pour 2020 devrait générer 50 téraoctets de données analysées par jour, 7000 téraoctets de données brutes par secondes. Par ailleurs, le volume est une caractéristique relative car ce qui était trop volumineux hier peut être considéré comme acceptable aujourd'hui.

Vélocité - Cette notion renvoie à la fréquence à laquelle sont générées, capturées, partagées et mise à jour les données, avec les contraintes que des flux croissants de données doivent être analysés en temps réel (ex : en bourse), de l'importance de la scalabilité (passage à l'échelle, capacité à monter en charge) des méthodes de traitement informatique.

Variété - Les données sont brutes, semi-structurées voire non structurées (ex : données provenant du web, composées de texte, d'images, de videos, de metadonnées, de code informatique,...). Il y a donc un besoin de nettoyage et de structuration pour le traitement des données.

Pour l'analyse de données massives, il se pose alors immédiatemennt le problème de la distribution des ressources informatiques. D'une part, on trouve la question du partage des ressources de calcul (calcul distribué ou parallèle). Le but est de répartir un calcul sur plusieurs entités (coeurs de processeurs ou processeurs). Ces entités peuvent être sur un même ordinateur (ex : supercalculateur) ou sur plusieurs ordinateurs (cluster de calcul). Par ailleurs, on est également confronté au partage des ressources de stockage des données (mémoire partagée/mémoire distribuée).

Dans le cas du Big Data, la solution retenue est très souvent un cluster de calcul avec une mémoire distribuée. Il s'agit d'une problématique très différente des codes de simulations numériques (météo par exemple). L'intérêt de ce type d'approche est que faire coopérer des machines simples est aussi puissant que développer un gros serveur. Ceci entrapine une réduction des coûts, une meilleure tolérance aux pannes (résilience), et la possibilité du passage à l'échelle (scalabilité).



Exemple de cluster. Source : programmation MapReduce sous R (R. Rakotomalala).

2 Hadoop

Hadoop est un logiciel “open source” de la fondation Apache. C’est un environnement logiciel dédié au stockage et au traitement de larges volumes de données. Hadoop repose sur deux composantes essentielles :

- Un système de fichiers distribué (HDFS : Hadoop Distributed File System) : permet de manipuler les données comme si elles étaient dans un seul fichier.
- Une implémentation efficace de l’algorithme Mapreduce : permet de traiter les données en parallélisant les calculs.

L’utilisateur gère le calcul et la mémoire comme s’il n’avait affaire qu’à un seul ordinateur. C’est un logiciel créé par Google qui est largement utilisé par Yahoo!, Facebook,...

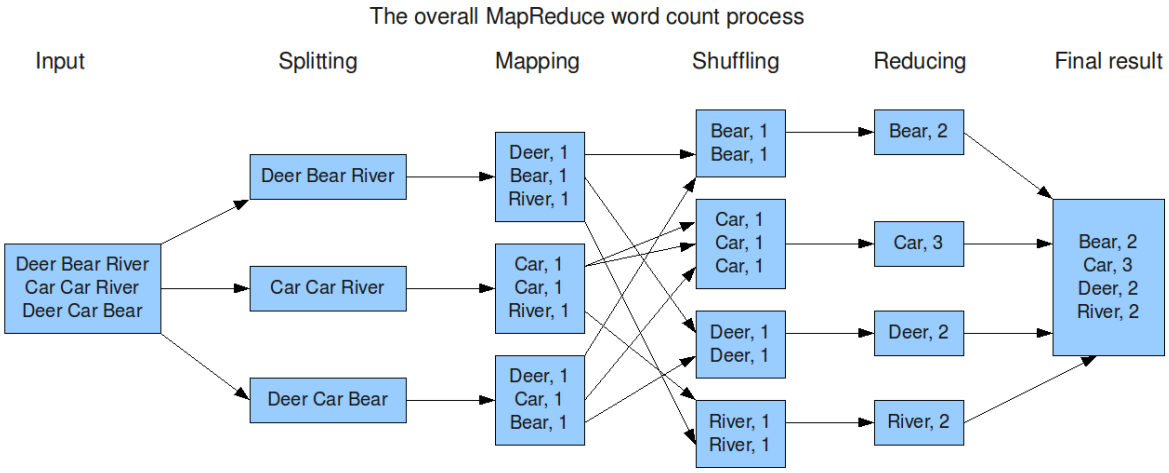
Quelques remarques importantes sur Le format HDFS sont les suivantes.

- Les données ne sont pas structurées et sont stockées sous forme de paires (clé,valeur) : un morceau de données correspond à une valeur et à chaque valeur est associé une clé.
- Des surcouches existent pour traiter des données structurées : Hbase (base de donnée orientée colonne), Hive/Pig langage pour interroger une base Hadoop avec une syntaxe de type SQL.
- Les données sont découpées en blocs par le noeud maitre et sont réparties sur le cluster; la taille d’un bloc est de 64Mb ou 128Mb.
- HDFS est tolérant aux pannes : les blocs sont répliqués trois fois; si un des éléments du cluster tombe en panne, le cluster s’adapte.
- La gestion de l’emplacement des différents blocs et de leurs répliques est géré par le NameNode. Tout cette gestion est automatisée et transparente pour l’utilisateur. C’est le NameNode qui répartie les taches de calcul en fonction des charges de travail de chaque noeuds de calcul (i.e. chaque partie du cluster).

Le paradigme MapReduce est un modèle de programmation permettant de programmer simplement des calculs distribués dans une architecture Hadoop. Une version (très simplifiée) d’un programme de type mapreduce est la suivante :

mapreduce(input,map,reduce)

- Les données (input) sont au format HDFS, elles sont donc découpées en blocs, dispatchées sur plusieurs machines.
- Sur chaque bloc est appliquée la fonction map qui prend en entrée des paires (clé,valeur) et renvoie des paires (clé,valeur).
- Les résultats sont remontés au niveau du noeud principal et regroupés par clés (shuffle).
- Pour chaque groupe de données possédant la même clé est appelé la fonction reduce() qui prend en entrée des paires (clé,valeur) et renvoie des paires (clé,valeur).



Exemple du comptage de mots. (Source : http://www.milanor.net/blog)

Pour optimiser les calculs on peut ajouter une étape “Compose” qui permet d’appliquer des traitement du type reduce() à des sous parties triées des données. Attention, l’étape Compose n’est pas forcément utilisée par Hadoop (dépend de la situation du cluster, bande passante,. . .).

3 RHadoop et la librairie rmr2

L’un des objectifs principaux de ce document est de familiariser le statisticien non-initié au paradigme MapReduce. Il existe à l’heure actuelle de nombreuses technologie concurrentes pour traiter de gros jeux de données mais pour éviter les coûts d’entrée importants en terme de connaissances nous nous restreignons à l’utilisation de la librairie `rmr2` de RHadoop. Cette dernière possède l’avantage d’être simple d’accès car elle repose sur le langage R et ne nécessite pas l’installation d’un serveur Hadoop. Les exemples traités dans cette partie s’inspirent très largement du TP disponible sur le site WikiStat[1] qui s’inspire lui même très largement du tuteuriel proposé par RHadoop[2].

3.1 Installation de rmr2

La librairie `rmr2` n’est pas disponible sur le CRAN, il faut donc réaliser l’installation à la main. Téléchargez la dernière version de `rmr2` sur l’espace collaboratif *GitHub* et l’installer. Par exemple pour Linux il suffit d’aller dans le dossier built et de taper la commande

```
R CMD INSTALL rmr2_3.3.0.tar.gz
```

Pour windows il faut utiliser le paquet `rmr2_3.3.0.zip`. Le chargement de `rmr2` dépend d’autres librairies dont l’installation ne pose pas de problème à l’exception de `rjava` qui peut s’avérer compliquée sous Windows.

[1] http://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-tutor5-R-mapreduce.pdf
[2] https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md

3.2 Fonctions

Comme d'habitude il conviendra d'utiliser l'aide en ligne pour avoir une description détaillée des différentes fonctions. Tout d'abord les données stockées au format HDFS sont du type `big.data.object`. Voici quelques fonctions qui seront utiles pour la suite.

`keyval` crée des paires (clé,valeur). `values` et `keys` permet d'extraire les clés et les valeurs de ces paires.

`to.dfs(kv)` où `kv` est une liste de (clé,valeur) ou un objet R (dans ce cas la clé est `NULL`) permet de transformer `kv` en un objet `big.data.object`. Par défaut cet objet est déposé dans un fichier temporaire mais il est possible de spécifier le chemin d'un fichier existant.

`from.dfs(input)` permet de transformer l'object big data `input` en un objet R de type liste de (clé,valeur) en mémoire.

La fonction `mapreduce` permet d'appliquer un traitement MapReduce à des données. Son fonctionnement précis sera détaillé par la suite. Par défaut `map` est la fonction identité , `combine` et `reduce` sont `NULL`.

4 Exemples élémentaires de programmation MapReduce

4.1 *Ab Initio*

On commence par charger la librarie `rmr2` et spécifier l'utilisation en locale, i.e. sans serveur Hadoop.

```
library(rmr2)
rmr.options(backend = "local")
```

Exécuter les instructions suivantes en prenant soin de bien identifier les types d'objets manipulés.

```
# création d'un objet de type big data
test <- to.dfs(1:10)
# retour à R
test2 <- from.dfs(test)
# création d'une liste de (clef,valeur)
test3 <- keyval(1,1:10)
keys(test3)
values(test3)
# mtcars est un data frame contenant la variable
# nombre de cylindres. Cette variables est définie comme
# clé, la valeur associée est la ligne correspondante du data frame
keyval(mtcars[,"cyl"],mtcars)
```

Il faut bien comprendre que la fonction `to.dfs` n'est utile que pour une utilisation locale de `rmr2`. Pour une utilisation réelle, on utilisera des fichier hdfs préexistants.

On donne maintenant deux exemples d'utilisation de la fonction `mapreduce`. Le premier exemple consiste à calculer des carrées d'entiers.

```
# carrés d'entiers
entiers <- to.dfs(1:10)
calcul.map = function(k,v){
  cbind(v,v^2)
}
calcul <- mapreduce(
  input = entiers,
  map = calcul.map
  # la fonction reduce est nulle par defaut
  )
resultat <- from.dfs(calcul)
resultat
```

Le deuxième exemple consiste à calculer la somme de carrés d'entiers.

```
calcul2.map = function(k,v){
  keyval(1,v^2)
}
calcul2.reduce = function(k,v){
  sum(v)
}
calcul2 <- mapreduce(
  input = entiers,
  map = calcul2.map,
  reduce = calcul2.reduce)
resultat2 <- from.dfs(calcul2)
resultat2
```

Un dernier exemple pour la route.

```
entiers3 <- to.dfs(1:1000000)
calcul3.map = function(k,v){
  keyval(1,1)
}
calcul3.reduce = function(k,v){
  sum(v)
}
calcul3 <- mapreduce(
  input = entiers3,
  map = calcul3.map,
  reduce = calcul3.reduce
  )
resultat3 <- from.dfs(calcul3)
resultat3
```

Question 1 *Comment expliquez-vous la valeur de* `resultat3` *?*

4.2 Structuration des données en blocs

Les exemples suivants permettent de visualiser tout d'abord le découpage en blocs d'un vecteur, puis d'une matrice de grande taille.

```
entiers4 <- to.dfs(1:1000000)
calcul4.map = function(k,v){
  keyval(v[1],v[length(v)])
}
calcul4.reduce = function(k,v){
  keyval(k,v)
}
calcul4 <- mapreduce(
  input = entiers4,
  map = calcul4.map,
  reduce = calcul4.reduce
  )
resultat4 <- from.dfs(calcul4)
resultat4
```

```
matrice6 <-to.dfs(matrix(1:1000000,10,100000))
calcul6.map = function(k,v){
  keyval(v[1,1],dim(v))
}
calcul6.reduce = function(k,v){
  keyval(k,v)
}
calcul6 <- mapreduce(
  input = matrice6,
  map = calcul6.map,
  reduce = calcul6.reduce
  )
resultat6 <- from.dfs(calcul6)
resultat6
```

```
matrice7 <-to.dfs(matrix(1:1000000,100000,10))
calcul7.map = function(k,v){
  keyval(v[1,1],dim(v))
}
calcul7.reduce = function(k,v){
  keyval(k,v)
}
calcul7 <- mapreduce(
  input = matrice7,
  map = calcul7.map,
  reduce = calcul7.reduce
  )
```

```
resultat7 <- from.dfs(calcul7)
resultat7
```

Question 2 *Comment expliquez-vous les valeurs de* `resultat4`, `resultat6` *et* `resultat7` *?*

On peut enfin regarder un dernier exemple où un objet R de type `list` est converti en objet HDFS.

```
matrice8 <-to.dfs(list(matrix(1:1000000,1000000,1),matrix(1:10,2,5),matrix(1:10,5,2)))
calcul8.map = function(k,v){
  keyval(v[[1]][1],dim(v[[1]]))
}
calcul8.reduce = function(k,v){
  keyval(k,v)
}
calcul8 <- mapreduce(
  input = matrice8,
  map = calcul8.map,
  reduce = calcul8.reduce
  )
resultat8 <- from.dfs(calcul8)
resultat8
```

Il est peut être remarqué que le premier élément de la liste est un vecteur de grande taille dont les éléments sont stockés sur un seul bloc... ce qui n'était pas le cas de l'exemple précédent (cf. `resultat4`).

4.3 Comptage d'entiers

Il s'agit de compter les nombres d'occurence de 50 tirages d'une loi de Bernouilli de paramètres 32 et 0,4. La fonction `tapply` le réalise en une seule ligne mais c'est encore un exemple didactique illustrant l'utilisation du paradigme mapreduce.

```
tirage <- to.dfs(rbinom(32,n=50,prob=0.4))
# le map associe à chaque entier une paire (entier,1)
comptage.map = function(k,v){
  keyval(v,1)
}
comptage.reduce = function(k,v){
  keyval(k,length(v))
}
comptage <- mapreduce(
  input = tirage,
  map = comptage.map,
  reduce = comptage.reduce)
from.dfs(comptage)
table(values(from.dfs(tirage)))
```

4.4 *Salve, munde!*

Il est temps maintenant de présenter l'exemple canonique (*hello world*) de MapReduce qui consiste à compter les mots d'un texte. Le principe est le même que pour le comptage d'entiers, à la différence près que l'étape map requiert plus de travail préparatoire pour découper le texte en mots.

```
#On définit une fonction wordcount pour le comptage de mots
wordcount = function(input,pattern = " "){
  #input : texte à analyser au format big data
  #pattern : sigle utilisé pour la séparation des mots
  #          (" " par défaut)
  wordcount.map = function(k,texte){
    keyval(unlist(strsplit(x = texte, split = pattern)),1)
  }
  wordcount.reduce = function(word,count){
    keyval(word, sum(count))
  }
  resultat<-mapreduce(
    input = input,
    map = wordcount.map,
    reduce = wordcount.reduce)
  return(resultat)
}

#Un exemple d'utilisation avec un texte simple
texte = c("un petit texte pour l'homme mais un grand grand grand texte pour l'humanité")
from.dfs(wordcount(to.dfs(texte)))
```

5 Quelques méthodes statistiques plus avancées

5.1 k-means

5.1.1 Principe

On rappelle succintement le principe du k-means.

- On a n points de $\mathbb{R}^d$ pour lesquels on cherche k classes en faisant r itérations.
- On initialise en affectant à chaque point une classe aléatoirement (uniformément sur $\{1, ..., k\}$).
- On calcule les barycentres de chaque classe.
- On reaffecte une classe à chaque point en déterminant le barycentre le plus proche puis on recalcule les barycentres de chaque classe. On itère $r - 1$ fois cette étape.

Dans le cadre MapReduce, on va itérer r fois une fonction `mapreduce`.

- La première étape ‘map‘ consiste à initialiser les affectations de classes. Les étapes ‘map‘ suivantes gèrent les affectations de classes.
- Les étapes ‘reduce‘ consistent à calculer les barycentres de chaque classes.

5.1.2 Programme

```
#fonction principale
kmeans = function(Pts,nbClusters,nbIter){
  #P : pts (au format big data) sur lesquels on fait un k-mean
  # nbClusters : nombre de clusters désirés
  # nbIter : nombre d'itérations dans le k-mean
  # retourne C : matrice des pts du k-mean

  #calcul de distance
  distance = function(c,P){
    #determine pour chaque point de P
    #la distance a chaque point de c
    resultat <- matrix(rep(0,nrow(P)*nrow(c)), nrow(P), nrow(c))
    for (i in 1:nrow(c)){
      resultat[,i] <- rowSums(
        (P-matrix(rep(c[i,],nrow(P)),nrow(P), ncol(P), byrow = TRUE))^2)
    }
    return(resultat)
  }
  #fonctions map : initialisation ou détermination des
  # centres les plus proches
  km.map = function(.,P){
    if (is.null(C)){
      #initialisation au premier tour
      pproche <- sample(1:nbClusters,nrow(P),replace = TRUE)
    }
    else {
      #sinon on cherche le plus proche
      D <- distance(C,P)
      pproche <- max.col(-D)
    }
    keyval(pproche,P)
  }
  #fonction reduce : calcul des barycentres
  km.reduce = function(.,G){
    t(as.matrix( apply(G,2,mean) ))
  }
  #programme principal
  # initialisation
  C <- NULL
  # iterations
```

```
  for(i in 1:nbIter){
    resultat = from.dfs(mapreduce(
      input = Pts,
      map = km.map,
      reduce = km.reduce))
    C <- values(resultat)
    # si des centres ont disparu
    # on en remet aléatoirement
    if (nrow(C) < nbClusters){
      C = rbind(C,matrix( rnorm((nbClusters-nrow(C))*nrow(C)),
                          ncol = nrow(C))%*%C)
    }
  }
  return(C)
}
```

On peut tester le résultat sur des données simulées.

```
#simulation de 5 centres plus bruit gaussien
#on reinitialise le generateur pour faciliter le debogage
set.seed(1)
P <- matrix(rep(0,200),100,2)
for (i in 1:5){
  P[((i-1)*20+1):(i*20),] <- (matrix(rep(rnorm(2, sd = 20),20),ncol=2,byrow = TRUE) +
                                matrix(rnorm(40),ncol=2))
}
#test
resultat <- kmeans(to.dfs(P),5,8)
plot(P)
points(resultat,col="blue",pch=16)
```

Question 3 *En pratique l'algorithme* `kmeans` *donné précédemment ne peut pas fonctionner correctement sur de gros volumes de données et nécessite l'ajout de calculs pour diminuer la taille des données "remontées et l'ajout éventuel d'une étape "combine". Expliquez pourquoi et modifiez le code afin d'incorporer cette nouvelle étape.*

5.1.3 Un programme plus réaliste

Comme cela est dit dans la question précédente, l'algorithme `kmeans` donné ne peut pas fonctionner correctement sur de gros volumes de données. La raison est que la totalité des données est remontée jusqu'au noeud principal où les calculs de moyennes sont réalisés. Il est évident que pour calculer une moyenne, il est équivalent d'avoir à sa disposition la totalité des données ou bien des sommes partielles ainsi que le nombre d'éléments sommés. Le but est donc, dans les étapes "map" et "reduce", de sommer directement les données par centres, en conservant à chaque fois le nombre d'éléments sommés. Notons que cela permet de rajouter une étape "combine" qui, si besoin, vient réduire encore plus les données transitant entre les noeuds du réseau. Rappelons qu'une étape

combine permet d'appliquer une étape "reduce" seulement sur une sous partie des blocs. Le calcul des moyennes donnant les nouveaux centres est relégué après le mapreduce en divisant juste les sommes obtenues par les nombres d'éléments sommés.

```
#fonction principale
kmeans2 = function(Pts,nbClusters,nbIter){
  #P : pts (au format big data) sur lesquels on fait un k-mean
  # nbClusters : nombre de clusters désirés
  # nbIter : nombre d'itérations dans le k-mean
  # retourne C : matrice des pts du k-mean

  #calcul de distance
  distance = function(c,P){
    #determine pour chaque point de P
    #la distance a chaque point de c
    resultat <- matrix(rep(0,nrow(P)*nrow(c)), nrow(P), nrow(c))
    for (i in 1:nrow(c)){
      resultat[,i] <- rowSums(
        (P-matrix(rep(c[i,],nrow(P)),nrow(P), ncol(P), byrow = TRUE))^2)
    }
    return(resultat)
  }
  #fonctions map : initialisation ou détermination des
  # centres les plus proches
  km.map = function(.,P){
    if (is.null(C)){
      #initialisation au premier tour
      pproche <- sample(1:nbClusters,nrow(P),replace = TRUE)
    }
    else {
      #sinon on cherche le plus proche
      D <- distance(C,P)
      pproche <- max.col(-D)
    }
    #On ajoute un poids 1 à chaque points
    keyval(pproche,cbind(1,P))
  }


  #fonction reduce : calcul des barycentres
  km.reduce = function(k,G){
    keyval( k , t(as.matrix(apply(G,2,sum))) )
  }
  #programme principal
  # initialisation
  C <- NULL
  # iterations
  for(i in 1:nbIter){
```

```
    resultat = from.dfs(mapreduce(
      input = Pts,
      map = km.map,
      reduce = km.reduce,
      combine = TRUE,
      in.memory.combine = TRUE))
    C <- values(resultat)
    # calcul des moyennes : on divise les sommes par le nombre d'éléments sommés
    C <- C[,-1]/C[,1]
    # si des centres ont disparu
    # on en remet aléatoirement
    if (nrow(C) < nbClusters){
      C = rbind(C,matrix( rnorm((nbClusters-nrow(C))*nrow(C)),
                          ncol = nrow(C))%*%C)
    }
  }
  return(C)
}
```

On peut tester le résultat sur des données simulées.

```
#simulation de 5 centres plus bruit gaussien
#on reinitialise le generateur pour faciliter le debogage
set.seed(1)
P <- matrix(rep(0,200),100,2)
for (i in 1:5){
  P[((i-1)*20+1):(i*20),] <- (matrix(rep(rnorm(2, sd = 20),20),ncol=2,byrow = TRUE) +
                                matrix(rnorm(40),ncol=2))
}
#test
resultat <- kmeans2(to.dfs(P),5,8)
plot(P)
points(resultat,col="blue",pch=16)
```

5.2 Régression linéaire

5.2.1 Principe

On considère un modèle linéaire du type $Y = X\theta + \varepsilon$ avec ε un bruit, Y de taille $p \times 1$, X de taille $p \times k$, θ de taille $p \times 1$ et on suppose que k est "petit" et p très grand. L'estimateur des moindre carrés de θ est donné par $(X'X)^{-1}X'Y$. On va utiliser le principe map reduce pour calculer dans un premier temps $X'X$ et $X'Y$ qui ont le bon gout d'avoir des tailles raisonnables puis on termine le calcul sous R pour calculer $(X'X)^{-1}X'Y$.

5.2.2 Code

```
#fonction principale
regression = function(data){

  #fonction map pour XtX
  xtx.map = function(.,v){
    X <- as.matrix(cbind( matrix(rep(1,nrow(v)),ncol=1) , v[,-1] ))
    keyval(1, list(t(X)%*%X))
  }
  #fonction reduce pour XtX
  xtx.reduce = function(.,v){
    keyval(1,Reduce("+", v))
    #keyval(1,v)
  }
  #Calcul de XtX
  xtx = mapreduce(
    input = data,
    map = xtx.map,
    reduce = xtx.reduce)
  #fonction map pout Xty
  xty.map = function(.,v){
    y <- v[,1]
    X <- cbind(matrix(rep(1,nrow(v)),ncol=1) , v[,-1] )
    keyval(1, list(t(X)%*%y))
  }
  #Calcul de Xty
  xty = mapreduce(
    input = data,
    map = xty.map,
    reduce = xtx.reduce)
  #Regression linéaire
  xtx <- values(from.dfs(xtx))
  xty <- values(from.dfs(xty))
  return(solve(xtx,xty))
}
```

Essai sur un exemple.

```
#on reinitialise le generateur pour faciliter le debogage
set.seed(1)
#generation des variables explicatives
X <- matrix(rnorm(3000), ncol=10)
#generation des variables à expliquer
y <- X%*%( as.matrix(rep(c(1,2),5)) ) + matrix(rnorm(300),ncol=1)
# data frame
base <- data.frame("y"=y,X)
```

```
resultat1 <- regression(to.dfs(base))
resultat1
resultat2 <- lm(y~.,data=base)
resultat2
```

Une remarque pour terminer : si l'exemple n'est pas suffisament gros (200 lignes au lieu de 300 lignes par exemple) alors il n'y aura qu'une étape map et qu'une étape reduce ce qui peut créer des problèmes de debogage.

6 Exemples d'analyse d'un jeu de données réelles avec Rhadoop

6.1 Présentation des données

Nous avons compilé dans cette partie quelques erreurs classiques d'étudiants vues dans des projets rendus. Pour les illustrer nous nous basons sur un jeu de données de type graphe, représentant une petite sous-partie du réseau Internet en 2002. Le jeu de données est disponible sur la page

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

Commençons par une description succincte des données. Chaque site est numéroté et chaque lien d'un site pointant vers un autre site est représenté par un arc orienté. Le fichier `web-google.txt` est composé de lignes, chaque ligne codant un lien du réseau. Une ligne est donc composée de deux nombres indiquant le site de départ du lien et le site d'arrivée. Un exemple simple valant mieux qu'un long discours : la ligne `172 1304` signifie qu'il y a un lien sur la page `172` pointant vers la page `1304`.

Une fois le fichier téléchargé, on commence par charger les données dans R et les tranformer au format HDFS.

```
# Importation du jeu de données
web_google <- read.table("web-Google.txt", header=TRUE)

# On transforme l'ensemble de notre jeu de données en un objet "big.data.object"
web_google_bigdata <- to.dfs(web_google)
```

Pour mieux comprendre la structuration des données au format HDFS on peut faire quelques manipulations élémentaires.

```
#la fonction map fait remonter la taille de chaque bloc
bloc.map = function(k,v){
  keyval(1,dim(v))
}
#la fonction reduce ne fait rien
bloc.reduce = function(k,v){
  keyval(k,v)
}
calculbloc <- mapreduce(
```

```
  input = web_google_bigdata,
  map = bloc.map,
  reduce = bloc.reduce
  )
resultatbloc <- from.dfs(calculbloc)
#resultatbloc permet de voir le nombre de blocs
#et la taille de chaque bloc
resultatbloc
```

6.2 Nombre de liens

Nous allons maintenant compter le nombre de liens, ce qui revient tout simplement à compter le nombre de lignes du fichier. On commence par une première approche naive proposée par des étudiants.

```
# Transformation des données en big.data.object
temp <- web_google[,1]
web_google_bigdata2 <- to.dfs(temp)

arc.map = function(k,v){
  keyval(v,1)
  # La 'valeur' de notre paire vaut 1
}
arc <- mapreduce(
  input = web_google_bigdata2,
  map = arc.map)
# On somme toutes les 'valeurs' de nos paires, on obtiendra le nombre de ligne :
nombre_arc <- sum(values(from.dfs(arc)))
nombre_arc
```

Cette approche n'est pas bonne car elle traite les données en amont dans `R` : une seule colonne est extraite du fichier initial avant de convertir les données en HDFS. Cette approche n'est absolument pas envisageable lorsque l'on traite des vraies "Big Data" puisque, dans ce cas, les données sont directement accessible en HDFS et ne peuvent pas être chargées intégralement et transformées dans `R` à cause de la limitation de la mémoire vive.

On propose maintenant une seconde approche naive adaptée encore une fois de codes étudiants.

```
arc2.map = function(k,v){
  keyval(1,v[,1])
  # La clé vaut 1 et la valeur vaut la première colonne des données
}
arc2 <- mapreduce(
  input = web_google_bigdata,
  map = arc2.map)
#la clé 1 est reproduite autant de fois
#qu'il y a de lignes. On a juste à sommer
```

```
#pour obtenir le nombre total de lignes
nombre_arc2 <- sum(keys(from.dfs(arc2)))
nombre_arc2
```

Cette fois, les données ne sont pas traitées avant d'être converties en HDFS. Par contre, on se rend vite compte que toutes les données de la première colonne sont remontées jusqu'au noeud maître pour pouvoir ensuite compter la taille des données. Cette approche n'est absolument pas envisageable en pratique car elle fait remonter la moitié des données totales au noeud maître (concrètement, la première colonne du fichier). On peut remarquer que remonter une donnée brute (numéro d'un site) ou remonter un 0 revient au même, il est donc possible de modifier légèrement le code précédent pour diminuer la taille des données remontées.

```
arc3.map = function(k,v){
  keyval(1,0*v[,1])
  # La clé vaut 1 et la valeur vaut la première colonne des données
  # où les valeurs sont remplacées par des 0
}
arc3 <- mapreduce(
  input = web_google_bigdata,
  map = arc3.map)
#la clé 1 est reproduite autant de fois
#qu'il y a de lignes. On a juste à sommer
#pour obtenir le nombre total de lignes
nombre_arc3 <- sum(keys(from.dfs(arc3)))
nombre_arc3
```

En terme de quantité de données remontées c'est déjà meilleur mais on peut faire beaucoup mieux. En effet, il est beaucoup plus intéressant d'utiliser les capacités de calcul de chacun des noeuds pour pouvoir calculer le nombre de lignes sur chacun des blocs et ensuite remonter l'information. Cela donne en pratique le code suivant.

```
arc4.map = function(k,v){
  keyval(1,length(v[,1]))
  # La clé vaut 1 et la valeur vaut la longueur de la première colonne des données
}
arc4 <- mapreduce(
  input = web_google_bigdata,
  map = arc4.map)
#On a juste a sommer les valeurs
nombre_arc <- sum(values(from.dfs(arc4)))
nombre_arc
```

Il est important de remarquer que dans ce dernier exemple, la quantité de données remontées au noeud maître est bien plus faible que dans les trois premières solutions proposées. Notons également qu'il est possible de rajouter une étape de "combine" sans aucune difficulté supplémentaire.

6.3 Nombre d'arcs non orientés

Nous souhaitons maintenant calculer le nombre d'arcs non orientés dans le graphe, c'est à dire que l'on ne tient plus compte du sens des liens. Dit autrement, si deux sites se citent mutuellement, les deux liens (arcs orientés) comptent pour un seul arc non orienté. Compter le nombre d'arcs non orientés est donc équivalent à compter le nombre de paires de sites se citant mutuellement, modulo le calcul du nombre d'arcs orientés. Traiter cette question algorithmiquement n'est pas bien compliquée. Il suffit par exemple de réécrire chaque ligne du fichier en rangeant dans l'ordre croissant chaque paire : par exemple la ligne `1910 12` devient `12 1910` et la ligne `120 11222` reste identique. Puis on range les lignes dans l'ordre lexicographique. Enfin on supprime les doublons.

Si l'on souhaite maintenant traiter ce problème à l'aide du paradigme map-reduce, cela n'est pas possible a priori. En effet, le paradigme map-reduce suppose qu'il est possible de traiter des sous parties des données de manière indépendante avant d'appliquer le traitement final. Or dans le cadre du calcul du nombre d'arcs non orientés, il est nécessaire d'avoir une vision sur toutes les données afin de pouvoir dire si une paire apparait plusieurs fois. Si l'on applique le traitement algorithmique énoncé précédemment à chacun des blocs, il est possible que quelques doublons soient supprimés, si par chance les deux liens se trouvent dans le même bloc. Mais globalement la majorité des données seront remontées au noeud maître ce qui n'est pas acceptable en pratique lorsque l'on traite de gros jeux de données.

Cet exemple simple permet de voir que le paradigme map-reduce est très contraignant et ne permet pas forcément de faire tous les traitements algorithmiques que l'on souhaiterait, même certains traitements simples.