PACKAGE RMR2

RAPPEL

Avant de commencer, voici un bref rappel de ce que c'est que le **mapreduce**.

MapReduce est un modèle de programmation créé par Google pour le traitement et la génération de larges ensembles de données sur des clusters d'ordinateurs. Il s'agit d'un composant central du Framework logiciel Apache Hadoop, qui permet le traitement résilient et distribué d'ensembles de données non structurées massifs sur des clusters d'ordinateurs, au sein desquels chaque nœud possède son propre espace de stockage.

Chaque algorithme de traitement de données basé sur MapReduce doit contenir deux types de programmes:

- Les Mappers : Ce sont des fonctions qui reçoivent en entrée les données à traiter et produisent pour chaque ligne de données, un résultat sous forme d'une paire (clé,valeur).
- Les Reducers : Ces fonctions reçoivent les résultats des mappers et les réduisent au moyen de diverses sortes d'opérations afin d'en obtenir un résultat plus concis correspondant à celui attendu par l'utilisateur.

I- INTRODUCTION AU PACKAGE RMR2

Dans ce tutoriel, nous nous intéressons à la programmation de MapReduce en R. Nous utiliserons la technologie RHadoop de la société Revolution Analytics et en particulier le package "rmr2" qui permet d'apprendre la programmation de MapReduce sans avoir à installer l'environnement Hadoop. Le paquet rmr2 permet d'effectuer de gros traitements et analyses de données via MapReduce sur un cluster Hadoop.

Après avoir installer le package rmr2, nous présenterons dans un premier temps des exemples très simples de cas d'utilisations, puis, dans un deuxième temps, nous progresserons en programmant un algorithme simple d'exploration de données comme la régression linéaire multiple.

a- Pré-requis et installation

La librairie rmr2 n'est pas disponible sur le CRAN, il faut donc l'installer depuis github. Et pour cela nous allons nous servir du package devtools. Ce package vous permet d'installer des packages dans R-studio directement depuis github

- Tout d'abord, installez devtools

```
install.packages("devtools")
library(devtools)
```

- Ensuite installez rmr2

```
devtools::install_github(c('RevolutionAnalytics/rmr2/pkg'))
```

b- Fonctions utiles

Il conviendra d'utiliser l'aide en ligne pour avoir une description détaillée des différentes fonctions du package rmr2. Voici quelques fonctions qui seront utiles pour la suite.

- Lire ou écrire des objets "R" depuis ou vers le système de fichiers:

from.dfs, to.dfs

- Créer, projeter ou concaténer des paires clé-valeur:

keyval

- MapReduce en utilisant le streaming Hadoop:

mapreduce

- L'objet "big-data":

big.data.object

- Manipulation de fichiers:

dfs.empty

- Equijoins utilisant map-reduce:

equijoin

- Paramètres importants de Hadoop en relation avec la rmr2:

hadoop.settings

- Créer des combinaisons de paramètres pour des OI flexibles:

make.input.format

- Fonction permettant de définir et d'obtenir les options du paquet:

rmr.options

- Exemples de grands ensembles de données:

rmr.sample

- Afficher le contenu d'une variable:

rmr.str

- Fonctions permettant de diviser un fichier en plusieurs parties ou de fusionner plusieurs parties en une seule:

scatter

- Définir le statut et définir et incrémenter les compteurs pour un emploi Hadoop:

status

- Creer des fonctions map-reduce à partir d'autres fonctions:

to.map

II- APPLICATIONS ELEMENTAIRES

a- Ab Initio

On commence par charger la librarie rmr2 et spécifier l'utilisation en locale,c'est à dire sans serveur Hadoop. La deuxième commande nous donne la possibilité de pratiquer la programmation de MapReduce sans avoir à installer l'environnement Hadoop.

```
library(rmr2)
```

```
## Warning: S3 methods 'gorder.default', 'gorder.factor', 'gorder.data.frame',
## 'gorder.matrix', 'gorder.raw' were declared in NAMESPACE but not found
```

```
## Please review your hadoop settings. See help(hadoop.settings)
```

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

```
## NULL
```

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

- Lire ou écrire des objets "R" depuis ou vers le système de fichiers

```
# création d'un objet de type big data
test <- to.dfs(1:20)

# retour à R
test2 <- from.dfs(test)
test2
```

```
## $key
## NULL
##
## $val
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

La première ligne place les données dans le HDFS, où la majeure partie des données doit résider pour que mapreduce puisse fonctionner. Il n'est pas possible d'écrire du big data de manière évolutive avec "to.dfs". "to.dfs" est néanmoins très utile pour diverses utilisations comme l'écriture de cas de test, l'apprentissage et le débogage. "to.dfs" peut mettre les données dans un fichier de votre choix, mais si vous n'en spécifiez pas un, il créera des fichiers temporaires et les nettoiera une fois terminé. La valeur de retour est ce que nous appelons un objet big data. Vous pouvez l'assigner à des variables, la passer à d'autres fonctions rmr, cartographier des emplois ou la relire. C'est un talon, c'est-à-dire que les données ne sont pas en mémoire, mais seulement des informations qui aident à trouver et à gérer les données. De cette façon, vous pouvez vous référer à de très grands ensembles de données dont la taille dépasse les limites de la mémoire.

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.

"from.dfs" est complémentaire de "to.dfs" et renvoie une collection de paires de clés-valeurs. "from.dfs" est utile pour définir des algorithmes de mapreduce chaque fois qu'un mapreduce produit quelque chose de taille raisonnable, comme un résumé, qui peut tenir en mémoire et doit être inspecté pour décider des prochaines étapes, ou pour le visualiser. Il est beaucoup plus important que le "to.dfs" dans le travail de production.

- Créer des paires clé-valeur

```
# création d'une liste de (clef,valeur)
test3 <- keyval(1,1:20)
keys(test3)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
values(test3)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

La fonction keyval() renvoie les paires clé-valeur, c'est-à-dire qu'elle associe une clé à chaque valeur du vecteur d'entrée.

- Mapping clé valeur avec le dataset mtcars

```
# 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)
```

```
## $key
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
##
## $val
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
```

```
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Dans cet exemple, nous avons pris comme clés les valeurs présentes dans la colonne “cyl”.

- Utilisation du mapreduce

La fonction mapreduce() est essentielle. Il faut ici trois paramètres : **“input” est l’ensemble de données à traiter ;** “map” est la fonction appelée pour cartographier les données en paires clé-valeur ; **“reduce” traite le sous-ensemble de données et renvoie le résultat avec sa clé.

Voici un exemple d’utilisation de la fonction mapreduce. Il consiste à calculer des carrées d’entiers.

```
# carrés d'entiers
entiers <- to.dfs(1:10)
calcul.map = function(k,v){
  keyval(v,v^2)
}

# la fonction reduce est nulle par defaut
calcul <- mapreduce(input = entiers,map = calcul.map)

resultat <- from.dfs(calcul)
resultat
```

```
## $key
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $val
##  [1]   1   4   9  16  25  36  49  64  81 100
```

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
```

```
## $key
## NULL
##
## $val
## [1] 385
```

b- Comptage d'entiers

Il s'agit de compter les nombres d'occurence de 50 tirages d'une loi de Bernoulli 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)
```

```
## $key
##  [1] 13 16 17 14 15  9 11 10 12 19  8
##
## $val
##  [1] 9 2 3 6 4 5 9 4 6 1 1
```

```
table(values(from.dfs(tirage)))
```

```
##
##  8  9 10 11 12 13 14 15 16 17 19
##  1  5  4  9  6  9  6  4  2  3  1
```

c- Comptage de mots

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)))
```

```
## $key
## [1] "un"          "petit"       "texte"       "pour"        "l'homme"
## [6] "mais"        "grand"       "l'humanité'"
##
## $val
## [1] 2 1 2 2 1 1 3 1
```

III- APPLICATIONS AVANCEES

a- Transmission d'un dataframe

Dans cette section, nous voulons calculer la somme des carrés des résidus (sum of the squared residuals - SSR) pour une analyse unidirectionnelle de la variance (ANOVA). L'originalité ici réside dans la manipulation et la transmission d'une data frame aux nœuds. Il s'agira de subdiviser la data frame en plusieurs parties.

Préparation des données: Nous créons notre jeu de données comme suit :

```
#group membership of the individuals
y <- factor(c(1,1,2,1,2,3,1,2,3,2,1,1,2,3,3))
#values of the response variable
x <- c(0.2,0.65,0.8,0.7,0.85,0.78,1.6,0.7,1.2,1.1,0.4,0.7,0.6,1.7,0.15)
#create a data frame from y and x
don <- data.frame(cbind(y,x))
don
```

```
##    y    x
## 1  1 0.20
## 2  1 0.65
## 3  2 0.80
## 4  1 0.70
## 5  2 0.85
## 6  3 0.78
## 7  1 1.60
## 8  2 0.70
## 9  3 1.20
## 10 2 1.10
## 11 1 0.40
## 12 1 0.70
## 13 2 0.60
## 14 3 1.70
## 15 3 0.15
```

Ci dessus notre tableau de données (format interne data.frame)

Nous devons définir les procédures de map() et de reduce().

MAP: Y est une variable catégorielle, elle indique l'appartenance au groupe. Nous l'utilisons directement pour définir les éléments clés.

```
#map
map_ssq <- function(., v){
  #the column y is the key
  cle <- v$y
  #return key and the entire data frame
  return(keyval(cle,v))
}
```

La fonction renvoie l'objet dataframe avec la clé en utilisant la fonction keyval().

REDUCE: Le dataframe initial est subdivisé en sous-ensembles définis par Y. Nous calculons la somme des carrés à l'intérieur de chaque groupe. "v" est une partie du cadre de données "don" ici. Nous avons toutes les variables mais seulement une partie des lignes.

```
#reduce
reduce_ssq <- function(k,v){
  #print the subset of the data frame used
  print("reduce") ; print(v)
  #number of row of the data frame
  n <- nrow(v)
  # calculate the sum of squares
  ssq <- (n-1) * var(v$x)
  #return the key and the result (value)
  return(keyval(k,ssq))
}
```

"v" est un objet dataframe, nous utilisons nrow() et non length() pour obtenir le nombre de lignes. Nous utilisons aussi l'opérateur "$" pour lire la colonne "x". La fonction renvoie en sortie la clé et le résultat du calcul.

CALCUL: Nous appelons la procédure mapreduce() de "rmr2".

```
#rmr2 format
don.dfs <- to.dfs(don)

#mapreduce
calcul <- mapreduce(input=don.dfs,map=map_ssq,reduce=reduce_ssq)
```

```
## [1] "reduce"
##    y    x
## 1  1 0.20
## 2  1 0.65
## 4  1 0.70
## 7  1 1.60
## 11 1 0.40
## 12 1 0.70
## [1] "reduce"
##    y    x
```

```
## 3  2 0.80
## 5  2 0.85
## 8  2 0.70
## 10 2 1.10
## 13 2 0.60
## [1] "reduce"
##    y    x
## 6  3 0.78
## 9  3 1.20
## 14 3 1.70
## 15 3 0.15
```

```
#retrieve the result
resultat <- from.dfs(calcul)
print(resultat)
```

```
## $key
## [1] 1 2 3
##
## $val
## [1] 1.152083 0.142000 1.293675
```

Nous obtenons les somme des carrés des éarts à l'intérieur de chaque sous-population qui sont identifiées par l'élément clé. Nous effectuons ensuite la somme pour obtenir la somme des carrés des résidus.

```
#SSR
ssr <- sum(resultat$val)
print(ssr)
```

```
## [1] 2.587758
```

Nous obtenons la valeur SSR = 2.587758.

C'est le bon résultat que l'on peut obtenir avec la fonction AOV de R par exemple.

```
#contrôle
print(aov (x ~ y))
```

```
## Call:
##    aov(formula = x ~ y)
##
## Terms:
##                         y Residuals
## Sum of Squares  0.149015  2.587758
## Deg. of Freedom        2        12
##
## Residual standard error: 0.4643776
## Estimated effects may be unbalanced
```

b- Régression linéaire

Dans cette section. Les données sont découpées en 2 blocs par la fonction map() (avec 2 valeurs de clés distinctes – la généralisation à K blocs ne pose pas de problèmes, il suffit de modifier la fonction map).

Les calculs sont réalisés pour chaque bloc par la fonction reduce(). A la sortie, nous effectuons la consolidation en additionnant les matrices.

Nous profiterons également de cet exemple pour aller plus loin dans la manipulation des données. Plutôt que de renvoyer une valeur atomique à la sortie de la fonction reduce(), nous enverrons une structure un peu plus complexe. Nous pourrons ainsi évaluer la souplesse de l'outil lorsqu'il s'agit d'aller vers des traitements plus élaborés.

**Données: Nous utiliserons les données mtcars [data(mtcars)]. Nous cherchons à expliquer la consommation (mpg) en fonction des autres variables.

```
mtcars
```

```
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280           19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C          17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE         16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL         17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC        15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4  8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial  14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128           32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic        30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona      21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin        15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28         13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9          27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa       30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L     15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino       19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora      15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E         21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

**MAP: La stratégie map() consiste à subdiviser les données (le data frame) en plusieurs parties. Voici le code pour une partition aléatoire en 2 portions à peu près égales.

```
#map
map_lm <- function(., D){
  #génération de valeurs aléatoires
  alea <- runif(nrow(D))
  #clé - découpage en 2 parts à peu près égales
  #on peut facilement multiplier les sous-groupes
  cle <- ifelse(alea < 0.5, 1, 2)
  #renvoyer la clé et les données
  return(keyval(cle,D))
}
```

Remarque 1 : Le caractère aléatoire de la partition n'est pas obligatoire dans le contexte de la régression. Nous aurions pu tout aussi bien prendre les n1 premiers individus pour la 1ère portion et les n2 suivants pour la seconde (avec taille d'échantillon = n = n1 + n2). Par conséquent, si les fragments de données sont situés sur des machines différentes, il sera tout à fait possible d'effectuer les calculs localement avant de consolider les résultats.

Remarque 2 : La généralisation en une subdivision en K sous-groupes d'observations ne pose absolument aucun problème. Ainsi, le code reduce() et la consolidation qui suivent fonctionneront quel que soit le nombre de nœuds sollicités.

**REDUCE: Penchons-nous un peu sur l'estimation des moindres carrés ordinaires (MCO) avant de décrire la fonction reduce(). Le modèle s'écrit :

$$y = xA + \varepsilon$$

Y est la variable cible ; X est la matrice correspondant aux variables prédictives, une première colonne de valeurs 1 est accolée à la matrice pour tenir compte de la constante de la régression ; a est le vecteur des paramètres ; "e" est le terme d'erreur qui résume les insuffisances du modèle.

L'estimateur des moindres carrés ordinaires â est défini par la formule:

$$\hat{a} = (X^t X)^{-1} X^t y$$

Où Xt est la transposée de la matrice X.

Regardons de près les coefficients des matrices pour comprendre la décomposition des calculs. Pour (XtX), au croisement des variables Xj et Xm, nous avons :

$$\sum_{i=1}^{n} x_{ij} \times x_{im}$$

Les termes étant additifs, nous pouvons fractionner les calculs en 2 parties :

$$\sum_{i=1}^{n_1} x_{ij} \times x_{im} + \sum_{i=n_1+1}^{n} x_{ij} \times x_{im}$$

Il en est de même pour (Xty), au croisement de Xj et y:

$$\sum_{i=1}^{n} x_{ij} \times y_i = \sum_{i=1}^{n_1} x_{ij} \times y_i + \sum_{i=n_1+1}^{n} x_{ij} \times y_i$$

Subdiviser les calculs en K parties ne pose absolument aucun problème au regard de ces propriétés. Nous les exploitons (ces propriétés) pour écrire la fonction reduce() :

```
#reduce
reduce_lm <- function(k,D){
  #nombre de lignes
n <- nrow(D)
  #récupération de la cible
y <- D$mpg
  #prédictives
X <- as.matrix(D[,-1])
  #rajouter la constante en première colonne
X <- cbind(rep(1,n),X)
  #calcul de X'X
XtX <- t(X) %*% X
  #calcul de X'y
Xty <- t(X) %*% y
  #former une structure de liste
res <- list(XtX = XtX, Xty = Xty)
  #renvoyer le tout
  return(keyval(k,res))
}
```

La nouvelle subtilité est que nous utilisons une liste pour renvoyer les deux matrices (XtX) et (Xty). Il faudra être très attentif lorsqu'il faudra consolider les résultats pour former les matrices globales correspondantes.

**Calculs et récupération des résultats: Il ne reste plus qu'à la lancer les calculs...

```
#format rmr2
don.dfs <- to.dfs(mtcars)
#mapreduce
calcul <- mapreduce(input=don.dfs,map=map_lm,reduce=reduce_lm)
#récupération
resultat <- from.dfs(calcul)
print(resultat)
```

```
## $key
## [1] 1 1 2 2
##
## $val
## $val$XtX
##                     cyl       disp         hp       drat         wt       qsec
##        17.000   108.000    4017.10   2671.000    61.8100    54.4820   294.8700
## cyl   108.000   736.000   28686.00  18786.000   381.2400   371.8240  1850.1200
## disp 4017.100 28686.000 1197575.33 755455.700 13804.4070 14867.8643 68459.2140
## hp   2671.000 18786.000  755455.70 517319.000  9386.1700  9450.2040 44958.7300
## drat   61.810   381.240   13804.41   9386.170   230.2963   190.7326  1072.1136
## wt     54.482   371.824   14867.86   9450.204   190.7326   194.0888   941.1352
```

```
## qsec  294.870  1850.120   68459.21  44958.730  1072.1136   941.1352  5152.8283
## vs      6.000    28.000     795.20    561.000    23.4800    14.4480   112.6400
## am      8.000    44.000    1341.80   1141.000    32.7700    19.7030   134.1700
## gear   64.000   398.000   14428.10  10080.000   238.0800   196.9820  1099.7800
## carb   53.000   366.000   14132.20  10033.000   190.9000   183.1920   888.1400
##           vs       am      gear      carb
##        6.000    8.000    64.000    53.000
## cyl   28.000   44.000   398.000   366.000
## disp 795.200 1341.800 14428.100 14132.200
## hp   561.000 1141.000 10080.000 10033.000
## drat  23.480   32.770   238.080   190.900
## wt    14.448   19.703   196.982   183.192
## qsec 112.640  134.170  1099.780   888.140
## vs     6.000    3.000    23.000    11.000
## am     3.000    8.000    36.000    27.000
## gear  23.000   36.000   252.000   206.000
## carb  11.000   27.000   206.000   213.000
##
## $val$Xty
##           [,1]
##        342.700
## cyl   2013.600
## disp 69595.530
## hp   47649.200
## drat  1286.977
## wt    1000.758
## qsec  6009.806
## vs     155.300
## am     192.000
## gear  1326.500
## carb   966.600
##
## $val$XtX
##                   cyl      disp        hp       drat         wt       qsec
##        15.00    90.00   3366.00   2023.00    53.2800    48.4700   276.2900
## cyl    90.00   588.00  23186.40  13418.00   310.1600   307.5800  1625.4400
## disp 3366.00 23186.40 982052.14 535908.70 11290.3890 12223.6245 60342.2900
## hp   2023.00 13418.00 535908.70 316959.00  6986.1100  7021.5400 36133.4300
## drat   53.28   310.16  11290.39   6986.11   192.4944   167.9864   984.8004
## wt     48.47   307.58  12223.62   7021.54   167.9864   166.8123   886.9593
## qsec  276.29  1625.44  60342.29  36133.43   984.8004   886.9593  5140.6519
## vs      8.00    36.00   1059.20    718.00    30.5500    22.1100   158.0300
## am      5.00    22.00    524.10    508.00    19.8800    11.6400    91.5100
## gear   54.00   312.00  11222.20   7032.00   194.8700   169.6000   997.6800
## carb   37.00   238.00   9083.90   5743.00   130.3600   127.3100   659.5300
##           vs     am     gear    carb
##         8.00   5.00    54.00   37.00
## cyl    36.00  22.00   312.00  238.00
## disp 1059.20 524.10 11222.20 9083.90
## hp    718.00 508.00  7032.00 5743.00
## drat   30.55  19.88   194.87  130.36
## wt     22.11  11.64   169.60  127.31
## qsec  158.03  91.51   997.68  659.53
## vs      8.00   4.00    31.00   14.00
```

```
## am      4.00   5.00    21.00   11.00
## gear   31.00  21.00   200.00  136.00
## carb   14.00  11.00   136.00  121.00
##
## $val$Xty
##           [,1]
##        300.200
## cyl   1680.000
## disp 59109.550
## hp   36713.500
## drat  1093.300
## wt     908.995
## qsec  5604.939
## vs     188.500
## am     125.100
## gear  1110.400
## carb   675.300
```

Voyons en détail l'objet « résultat » :

Dans $key, nous disposons du vecteur (1, 1, 2, 2), nous remarquons que les clés se répètent 2 fois parce que notre fonction reduce() a retourné 2 éléments (XtX) et (Xty).

Dans $val, nous avons une structure de liste où les matrices (XtX) et (Xty) se succèdent pour chaque valeur de la clé. Pour former la matrice (XtX) globale [resp. (Xty)], il faudrait additionner les éléments en position (1, 3) [resp. (2, 4)].

**Consolidation des résultats: Les procédures de consolidation suivantes sont opérationnelles quel que soit le nombre de nœuds sollicités (c.-à-.d. nombre de clés K >= 1).

```
#consolidation
#X'X
MXtX <- matrix(0,nrow=ncol(mtcars),ncol=ncol(mtcars))
for (i in seq(1,length(resultat$val)-1,2)){
  MXtX <- MXtX + resultat$val[[i]]
}
print("MXtX")
```

```
## [1] "MXtX"
```

```
print(MXtX)
```

```
##                   cyl       disp         hp       drat         wt       qsec
##        32.000   198.000    7383.10    4694.00   115.0900   102.9520    571.160
## cyl   198.000  1324.000   51872.40   32204.00   691.4000   679.4040   3475.560
## disp 7383.100 51872.400 2179627.47 1291364.40 25094.7960 27091.4888 128801.504
## hp   4694.000 32204.000 1291364.40  834278.00 16372.2800 16471.7440  81092.160
## drat  115.090   691.400   25094.80   16372.28   422.7907   358.7190   2056.914
## wt    102.952   679.404   27091.49   16471.74   358.7190   360.9011   1828.095
## qsec  571.160  3475.560  128801.50   81092.16  2056.9140  1828.0946  10293.480
## vs     14.000    64.000    1854.40    1279.00    54.0300    36.5580    270.670
## am     13.000    66.000    1865.90    1649.00    52.6500    31.3430    225.680
## gear  118.000   710.000   25650.30   17112.00   432.9500   366.5820   2097.460
## carb   90.000   604.000   23216.10   15776.00   321.2600   310.5020   1547.670
```

```
##            vs       am      gear      carb
##        14.000   13.000   118.000    90.000
## cyl    64.000   66.000   710.000   604.000
## disp 1854.400 1865.900 25650.300 23216.100
## hp   1279.000 1649.000 17112.000 15776.000
## drat   54.030   52.650   432.950   321.260
## wt     36.558   31.343   366.582   310.502
## qsec  270.670  225.680  2097.460  1547.670
## vs     14.000    7.000    54.000    25.000
## am      7.000   13.000    57.000    38.000
## gear   54.000   57.000   452.000   342.000
## carb   25.000   38.000   342.000   334.000
```

```
#X'y
MXty <- matrix(0,nrow=ncol(mtcars),ncol=1)
for (i in seq(2,length(resultat$val),2)){
MXty <- MXty + resultat$val[[i]]
}
print("MXty")
```

```
## [1] "MXty"
```

```
print(MXty)
```

```
##            [,1]
##         642.900
## cyl    3693.600
## disp 128705.080
## hp    84362.700
## drat   2380.277
## wt     1909.753
## qsec  11614.745
## vs      343.800
## am      317.100
## gear   2436.900
## carb   1641.900
```

Nous obtenons les matrices globales (XtX) et (Xty)

**Estimation des paramètres de la régression: Les estimateurs â sont produits à l'aide de procédure solve() de R.

```
#coefficients de la régression
a.chapeau <- solve(MXtX,MXty)
print("A l'issue des calculs, les coefficients de la régression pour les données « mtcars » sont :")
```

```
## [1] "A l'issue des calculs, les coefficients de la régression pour les données « mtcars » sont :"
```

```
print(a.chapeau)
```

```
##             [,1]
```

```
##      12.30337416
## cyl  -0.11144048
## disp  0.01333524
## hp   -0.02148212
## drat  0.78711097
## wt   -3.71530393
## qsec  0.82104075
## vs    0.31776281
## am    2.52022689
## gear  0.65541302
## carb -0.19941925
```

**Vérification - Procédure lm() de R: A titre de vérification, nous avons effectué la régression à l'aide de la procédure lm() de R.

```r
#vérification
print(summary(lm(mpg ~ ., data = mtcars)))
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4506 -1.6044 -0.1196  1.2193  4.6271
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.30337   18.71788   0.657   0.5181
## cyl         -0.11144    1.04502  -0.107   0.9161
## disp         0.01334    0.01786   0.747   0.4635
## hp          -0.02148    0.02177  -0.987   0.3350
## drat         0.78711    1.63537   0.481   0.6353
## wt          -3.71530    1.89441  -1.961   0.0633 .
## qsec         0.82104    0.73084   1.123   0.2739
## vs           0.31776    2.10451   0.151   0.8814
## am           2.52023    2.05665   1.225   0.2340
## gear         0.65541    1.49326   0.439   0.6652
## carb        -0.19942    0.82875  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.65 on 21 degrees of freedom
## Multiple R-squared:  0.869,  Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

Les paramètres estimés concordent en tous points.

CONCLUSION

Des exemples très scolaires ont été mis en avant dans ce tutoriel pour illustrer la programmation MapReduce à l'aide du package « rmr2 » sous R. L'idée directrice est la subdivision des calculs sur un groupe (cluster) de machines (nœuds). Bien sûr, d'autres solutions existent.

Pour aller plus loin, il faudrait se placer sur une configuration où les données arrivent par blocs - par exemple en provenance de différentes machines - occasionnant plusieurs appels à la fonction map qui les réorganise avant de passer la main à la fonction reduce, qui peut être appelée plusieurs fois ou non selon le nombre de valeurs distinctes de la clé. Ceci serait possible par exemple si l'on travaillait dans un véritable environnement Hadoop avec un cluster à plusieurs nœuds.

RESSOURCES

- http://eric.univ-lyon2.fr/~ricco/tanagra/fichiers/en_Tanagra_MapReduce.pdf
- https://www.math.u-bordeaux.fr/~arichou/TP.pdf