Programmation MapReduce sous R

Ricco Rakotomalala

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

Plan de présentation

1. L'écosystème Hadoop
2. Principe de programmation MapReduce
3. Programmation des fonctions Map et Reduce avec RHadoop
4. Quelques exemples

HADOOP

Hadoop ?

Hadoop est un environnement logiciel « open source » de la fondation Apache. C'est un environnement logiciel dédié au stockage et au traitement distribués de larges volumes de données.

Hadoop repose sur deux composantes essentielles :

1. Un système de fichiers distribué (HDFS : hadoop distributed file system)
2. Une implémentation efficace de l'algorithme MapReduce

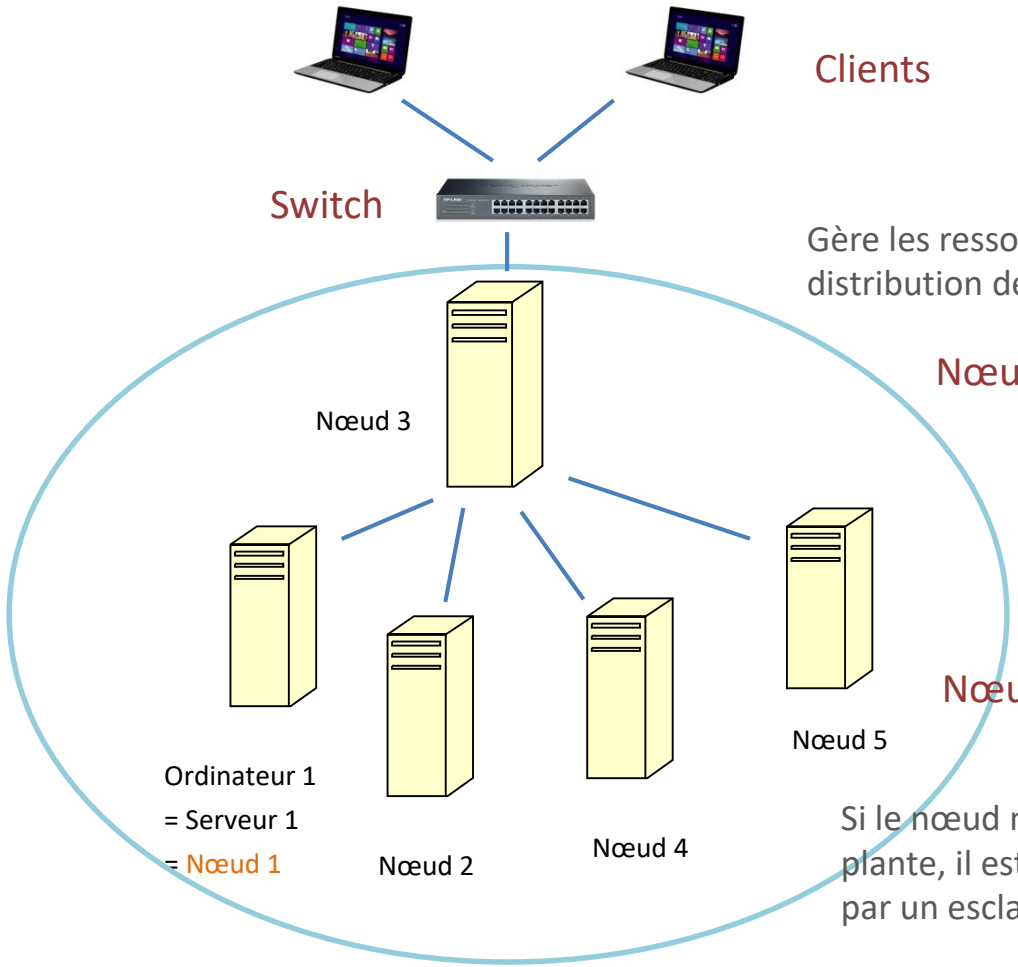
Outils Hadoop

Hadoop intègre des modules supplémentaires : Hadoop Common, qui est une bibliothèque d'utilitaires destinées à gérer les autres modules ; Hadoop YARN, qui gère les ressources et l'organisation des tâches (plus performante que la génération précédente, on l'associe à l'appellation ''MapReduce 2.0'' – Le moteur interne est amélioré, mais rien ne change pour nous programmeurs).

Hadoop propose une séries d'outils prêtes à l'emploi : HBase, une base de données non-relationnelle ; Hive, un entrepôts de données proposant un langage de requête proche du SQL ; Pig, une plate-forme haut-niveau qui permet de définir des programmes MapReduce à l'aide d'un langage spécifique similaire à SQL (Pig Latin) ; etc.

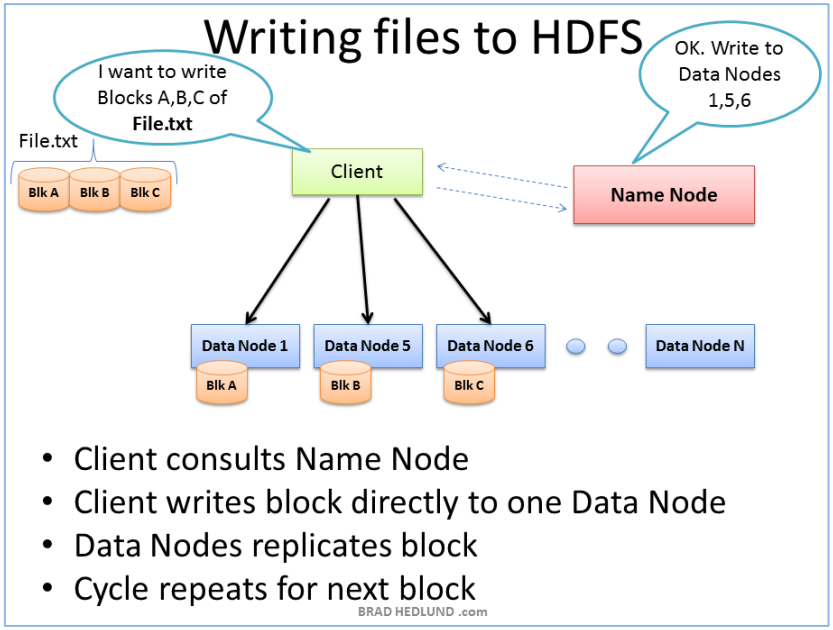
Idée : faire coopérer des machines simples est aussi puissant que développer un gros serveur

Intérêt : réduction des coûts, meilleure tolérance aux pannes (résilience), scalabilité (montée en charge)



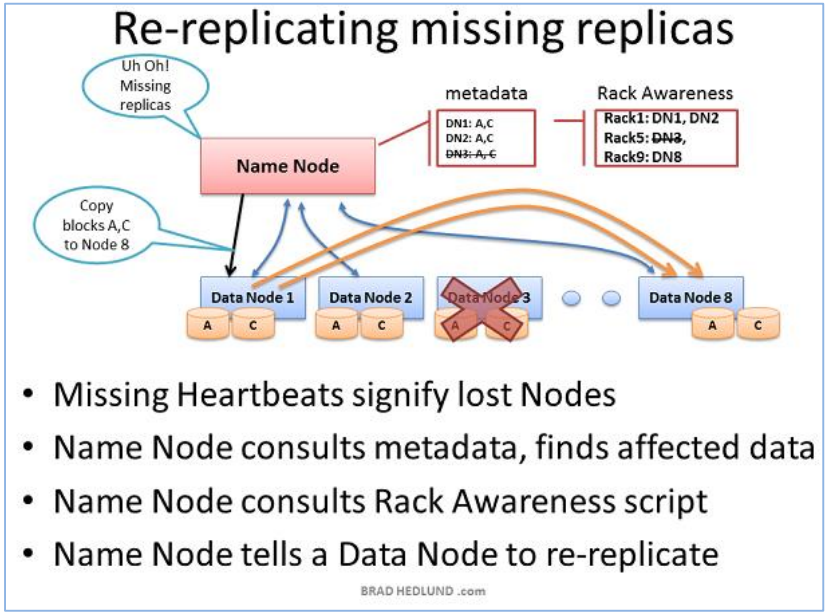
Cluster (grappe) de serveurs

Les nœuds sont groupés sous un même nom = nom du cluster qui permet de le désigner et le manipuler.



- Chaque fichier à stocker est subdivisé en blocs
- Chaque bloc est répliqué sur 3 nœuds

Si un des nœuds « tombe », les données sont automatiquement répliquées



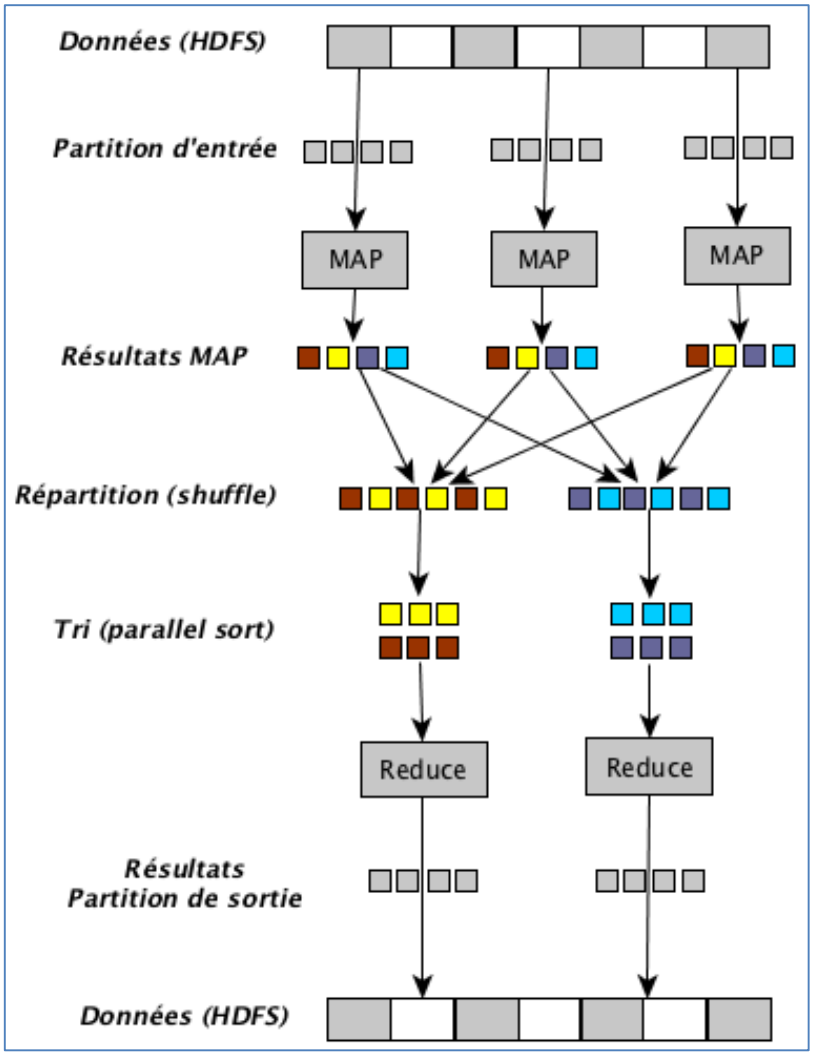
B. Hedlund, « Understanding Hadoop Clusters and the Network »
http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/

Modèle de programmation

MAPREDUCE

C. Parageaud, « MapReduce et les grilles de données, ou Hadoop sans Hadoop ».http://blog.ippon.fr/2014/03/14/mapreduce-et-les-grilles-de-donnees-ou-hadoop-sans-hadoop/

Idée : Définir un modèle de développement permettant de programmer « simplement » des calculs parallèles et distribués.



Accès aux données sur HDFS

Le système (Hadoop) se charge de splitter les données en blocs. On n'a pas prise là-dessus.

Sur chaque bloc est appliqué la fonction map() que nous devons programmer. Calculs + extraction des informations qui nous intéressent. Une liste d'objets indexés par une clé est produite en sortie.

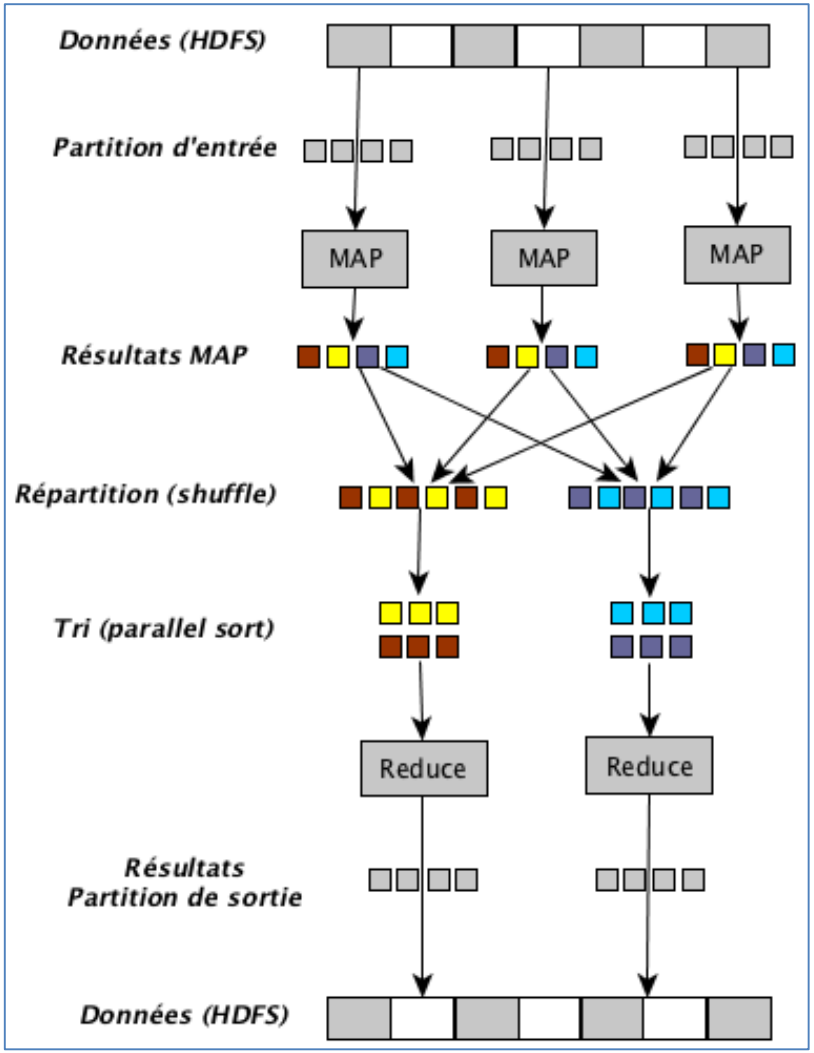
Le système réorganise les données de manière à regrouper les objets qui ont une clé identique.

Pour chaque groupe de données possédant la même clé est appelé la fonction reduce(). En sortie, nous avons une liste d'éléments calculés.

Les éléments sont écrits sur HDFS.

Notre travail consiste à programmer au mieux les fonction map() et reduce()

Objectif : Parallélisation des traitements par découpage des données en tranches.



Potentiellement, chaque MAP appelé peut s'exécuter sur un nœud du cluster.

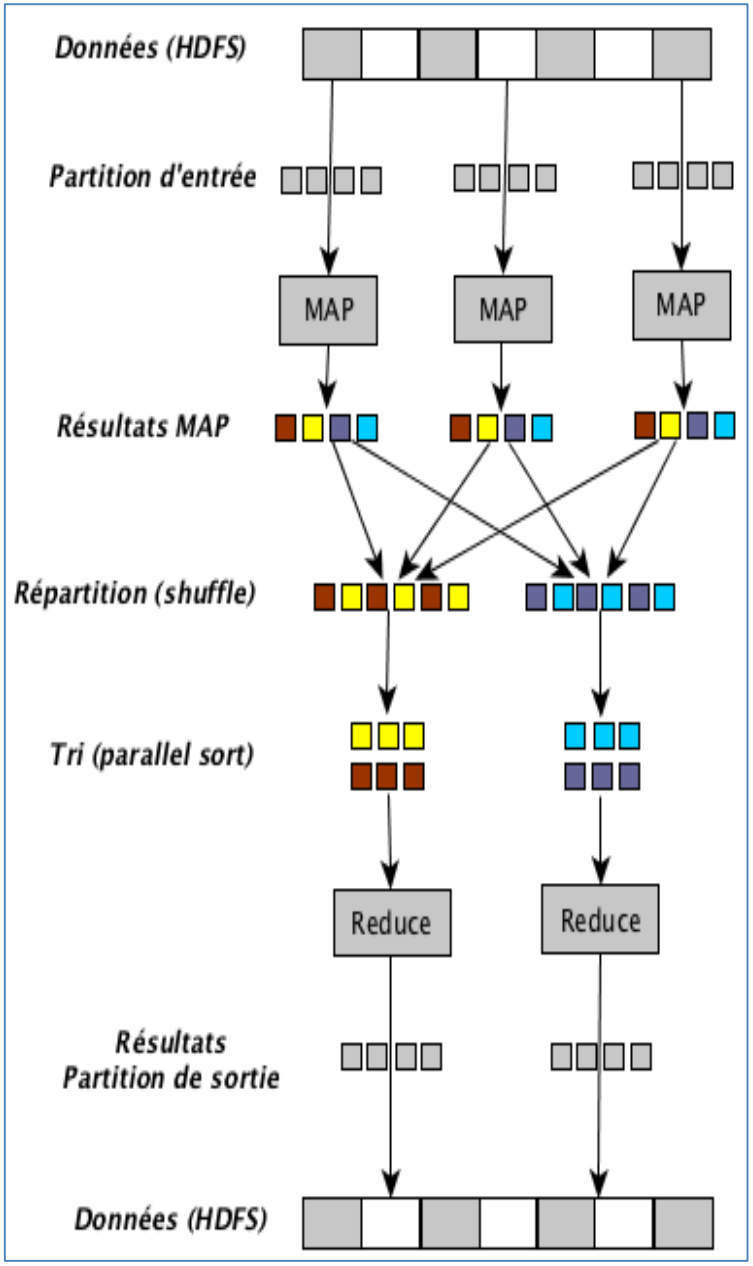
Si un nœud tombe en panne, la tâche est expédiée sur un nouveau nœud.

Goulot d'étranglement, la suite n'est possible que lorsque la dernière fonction MAP appelée a fini ses traitements.

Idem, chaque fonction REDUCE peut s'exécuter sur un nœud.

Remarque : s'il n'y a pas assez de nœuds, un système de file d'attente est mis en place, les nœuds rapides réaliseront plus de tâches.

MapReduce – Un exemple : comptage de lettres



Données : (AAABABBABBAAABBBAA)

Split en 3 blocs : (AAABAB) (BABBAA) (ABBBAA)
La fonction map() est appelée 3 fois.

La fonction map() associe à chaque lettre la valeur 1, et définit la lettre comme clé.

(AAABAB) (BABBAA) (ABBBAA)
(111111) (111111) (111111)

Le système répartit les données selon les valeurs de la clé. Nous avons 2 groupes de valeurs ici.

(AAAAAAAAAA) (BBBBBBBB)
(1111111111) (11111111)

A chaque groupe défini par la clé est appelée la fonction reduce() qui effectue la somme des ''1''. En sortie est fournie la clé et la somme calculée.

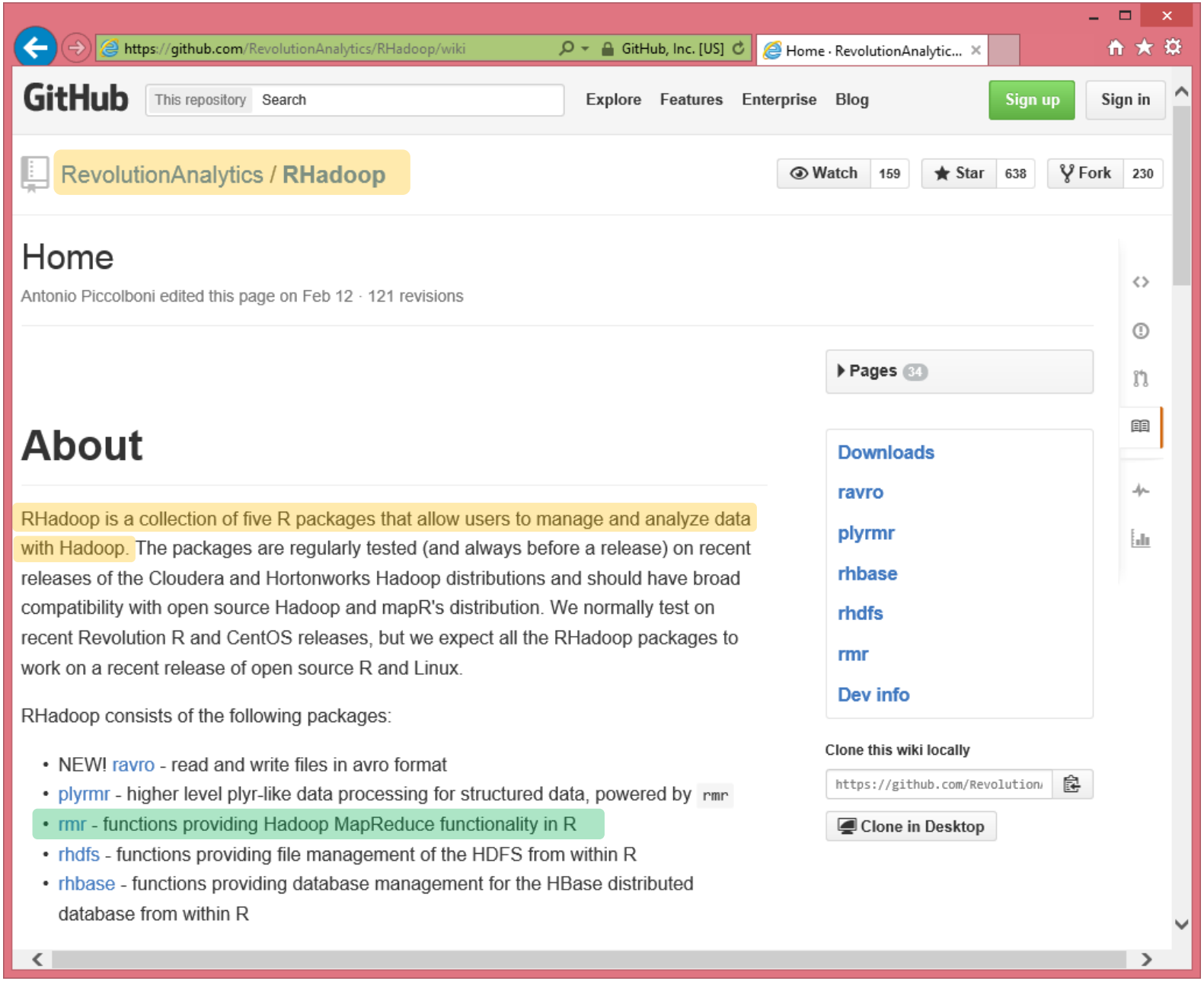
(A:10) (B:8)

Ces informations sont inscrites dans le fichier de sortie

Implémentation du principe MapReduce sous R

PROGRAMMATION SOUS R

Utilisation des packages de RHadoop



On peut s'initier à la programmation MapReduce sous R sans avoir à installer Hadoop (qui requiert d'autres compétences).

Il faut installer manuellement le package ''rmr'' et ses dépendances

to.dfs() transforme un objet de R (vecteur, data.frame, etc.) et l'écrit dans un fichier temporaire compatible HDFS

from.dfs() récupère le contenu d'un fichier temporaire HDFS et le transforme en un objet R

→ Ces fonctions ne sont pas nécessaires si nous travaillons réellement dans un environnement Hadoop

Tutoriel Tanagra, « MapReduce avec R », février 2015
http://tutoriels-data-mining.blogspot.fr/2015/02/mapreduce-avec-r.html

```
mapreduce(
input,
map,
reduce)
```

mapreduce est une fonction fournie par ‘’rmr’’ pour exécuter un job MapReduce

input représente les données à manipuler dans le job

map est la fonction Map() à programmer

reduce est la fonction Reduce() à programmer

La fonction renvoie un objet HDFS que l’on peut convertir en objet R avec la fonction from.dfs()

Remarque 1 : D’autres paramètres sont disponibles. Elles prennent des valeurs par défaut, nous ne sommes pas tenus de les modifier explicitement.
Remarque 2 : Le système s’occupe de la synchronisation. Tant que le dernier map() n’est pas fini, on ne passe pas à l’étape suivante.
Remarque 3 : Si un nœud plante, les traitements associés sont transférés sur un autre nœud.

Exemple de comptage de lettres

```
#chargement du package
library(rmr2)
#fonctionner sans Hadoop
rmr.options(backend="local")
#vecteur de lettres
lettres <- c("A","A","A","B","A","B","B","A","B","B","A","A","A","B","B","B","A","A")

#fonction map()
mon.map <- function(.,data){
  #créer un vecteur de valeur 1 de même longueur que v
  one <- rep(1,length(data))
  #l'associer à v qui joue le rôle de clé
  cle_valeur <- keyval(data,one)
  #renvoyer le tout
  return(cle_valeur)
}

#fonction reduce
mon.reduce <- function(k,v){
  #faire la somme de 1 pour une clé donnée
  somme <- sum(v)
  #renvoyer en sortie la clé et le résultat
  return(keyval(k,somme))
}

#transformation
dfs.lettres <- to.dfs(lettres)
#appel de mapreduce
res.mr <- mapreduce(input=dfs.lettres,map=mon.map,reduce=mon.reduce)
#conversion en R
resultat <- from.dfs(res.mr)
#affichage
print(resultat)
```

```
> #affichage
> print(resultat)
$key
[1] "A" "B"

$val
[1] 10  8
```

Comprendre le programme en ajoutant des affichages de contrôle - Map

Les données sont reçues d'un seul tenant dans notre exemple. Map() n'est appelé qu'une seule fois. Il peut y avoir ''split'' [et plusieurs appels de map()] quand le volume de données est élevé. C'est le système qui décide.

```
[1] "A" "A" "A" "B" "A" "B" "B" "A" "B" "B" "A" "A" "A" "B" "B" "B" "A" "A"
```

```
#fonction map()
mon.map <- function(.,data){
  #affichage des données en entrée
  print(data)
  #créer un vecteur de valeur 1 de même longueur que v
  one <- rep(1,length(data))
  #l'associer à v qui joue le rôle de clé
  cle_valeur <- keyval(data,one)
  #affichage des clés-valeurs
  print(cle_valeur)
  #renvoyer le tout
  return(cle_valeur)
}
```

Le vecteur de lettres va servir à définir les clés. Nous associons la valeur **1** à chaque lettre observée.

```
$key
 [1] "A" "A" "A" "B" "A" "B" "B" "A" "B" "B" "A" "A" "A" "B" "B" "B" "A" "A"

$val
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Il y a 2 valeurs de clés possibles (''A'' et ''B''). La fonction reduce() est appelée 2 fois.

```
#fonction reduce
mon.reduce <- function(k,v){
  #affichage de la clé
  print(k)
  #affichage des données associées
  print(v)
  #faire la somme de 1 pour une clé donnée
  somme <- sum(v)
  #clé + résulat
  cle.res <- keyval(k,somme)
  print(cle.res)
  #renvoyer en sortie la clé et le résultat
  return(cle.res)
}
```

```
[1] "A"
```

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

```
$key
[1] "A"

$val
[1] 10
```

1er appel pour la valeur de clé ''A''

''A''

```
[1] "B"
```

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

```
$key
[1] "B"

$val
[1] 8
```

2nd appel pour la valeur de clé ''B''

''B''

Remarque : Plutôt qu'un simple vecteur ($val est un vecteur à 1 valeur ici), on peut associer un objet plus complexe à la clé (ex. une matrice, une liste, etc.).

L'objet obtenu à la sortie est une liste avec deux champs $key et $val. Ils sont constitués de 2 vecteurs dans cet exemple, mais on peut aussi avoir des structures plus complexes. Tout dépend de ce que renvoie keyval() dans reduce().

```
> #affichage
> print(resultat)
$key
[1] "A" "B"

$val
[1] 10  8
```

Il est tout à fait possible de réaliser un post-traitement sur cet objet. Ex. détecter la lettre la plus fréquente.

```
#la lettre la plus fréquente
most.freq <- resultat$key[which.max(resultat$val)]
print(most.freq)
```

```
> print(most.freq)
[1] "A"
```

Pour pouvoir programmer en R sous Hadoop, il faut d'abord installer l'environnement. Dans le tutoriel « Programmation R sous Hadoop » (avril 2015), ont été installés successivement via la distribution CLOUDERA (un cluster simple nœud) :

1. Un logiciel de virtualisation permettant d'accueillir un système d'exploitation
2. Un système d'exploitation Linux (CentOS)
3. Hadoop est automatiquement installé, configuré et démarré
4. Le logiciel R
5. RStudio version serveur
6. Installer les packages nécessaires à RHadoop
7. Installer les packages de RHadoop
8. On peut enfin programmer et traiter des fichiers qui sont réellement sur HDFS

- VirtualBox a été utilisé
- CentOS = version gratuite de Red Hat
- On peut installer directement Apache Hadoop sans passer par ce dispositif (1)
- R va interpréter notre code
- Pour accéder à l'éditeur de manière distante via un navigateur
- La liste est connue et identifiée (ouf !)

Les affichages de contrôle ne sont plus possibles.

QUELQUES EXEMPLES

Construire une clé à partir de données : comptage des valeurs paires et impaires

```
#vecteur de valeurs entières
x <- c(2,6,67,85,7,9,4,21,78,45)

#map selon la parité de la valeur
map_valeurs <- function(., v){
 #calcul de la clé à partir des données
 cle <- ifelse (v %% 2 == 0, 1, 2)
 #retour des vecteurs clés et valeurs
 return(keyval(cle,v))
}

#reduce sur chaque sous-vecteur
reduce_valeurs <- function(k, v){
 #longueur du vecteur à traiter
 nb <- length(v)
 #renvoyer la clé et le résultat associé
 return(keyval(k, nb))
}

#transformation en type compatible rmr2
x.dfs <- to.dfs(x)
#fonction mapreduce de rmr2
calcul <- mapreduce(input = x.dfs, map = map_valeurs, reduce = reduce_valeurs)
#transformation en un type R usuel
resultat <- from.dfs(calcul)
#affichage
print(resultat)
```

La clé $key a été générée à partir des données $val

```
$key
 [1] 1 1 2 2 2 2 1 2 1 2

$val
 [1]  2  6 67 85  7  9  4 21 78 45
```

2 valeurs de clés ➔ 2 appels de la fonction reduce() avec les vecteurs ‘‘v’’ suivants

Clé = 1 : 2 6 4 78

Clé = 2 : 67 85 7 9 21 45

```
$key
[1] 1 2

$val
[1] 4 6
```

Traiter un data.frame : moyennes conditionnelles des rangs

```
#vecteur indicateur de groupe
y <- factor(c(1,1,2,1,2,3,1,2,3,2,1,1,2,3,3))
#vecteur de valeurs
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)
#construire un data frame
don <- data.frame(cbind(y,x))

# **** map - ''data'' est un data.frame ici ***
map.rank <- function(., data){
  #calculer les rangs des valeurs
  rang <- rank(data$x,ties.method="average")
  #la colonne y est la clé
  cle <- data$y
  #renvoyer la clé et le vecteur de rang
  return(keyval(cle,rang))
}

#*** reduce - calcul - ''v'' est un vecteur ici ***
reduce.mean <- function(k,v){
  #moyenne
  m <- mean(v)
  #renvoyer la clé et le résultat
  return(keyval(k,m))
}

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

Le data.frame n'est pas renvoyé en sortie (il est possible de le faire). La colonne ''x'' n'est plus utilisée par la suite.

```
$key
 [1] 1 1 2 1 2 3 1 2 3 2 1 1 2 3 3

$val
 [1]  2  5 10  7 11  9 14  7 13 12  3  7  4 15  1
```

3 valeurs de clé ➔ 3 appels de reduce() avec 3 vecteurs différents

Clé = 1 : 2 5 7 14 3 7

Clé = 2 : 10 11 7 12 4

Clé = 3 : 9 13 15 1

```
$key
[1] 1 2 3

$val
[1] 6.333333 8.800000 9.500000
```

De la documentation à profusion sur le net (**n'achetez jamais des livres sur R**)

Site du cours

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

Principe MapReduce

M. Dumoulin, « Introduction aux algorithmes MapReduce », février 2014.

Programmation MapReduce sous R

Hugh Devlin, "Mapreduce in R", janvier 2014.

Tutoriel Tanagra, "MapReduce avec R", février 2015.

Tutoriel Tanagra, "Programmation R sous Hadoop", avril 2015.

POLLS (Kdnuggets)

Data Mining / Analytics Tools Used? (May 2015)

(R, 2nd ou 1er depuis 2010)

What languages you used for data mining / data analysis? (July 2015)

(Langage R en 1ère position)