

BigData - Semaine 3

Pierre Nerzic

février-mars 2019

Le cours de cette semaine explique comment créer une application MapReduce pour YARN complète. Deux projets vont être expliqués. Ils concernent des statistiques sur des documents.

- calcul de la variance
- calcul de la valeur médiane

On va appliquer ça à la longueur des lignes de textes, mais ça pourrait être la durée de séjour sur une place de parking payante, la température mesurée à midi, le nombre de sangliers dans les forêts. . .

Application

Soient des données : des romans sous forme de fichiers texte. On s'intéresse à la longueur des lignes et on veut la variance...

1897

DRACULA

by Bram Stoker

CHAPTER I.

JONATHAN HARKER'S JOURNAL.

(Kept in shorthand.)

3 May. Bistriz.- Left Munich at 8:35 P.M., on 1st May, arriving at Vienna early next morning; should have arrived at 6:46, but train was an hour late. Buda-Pesth seems a wonderful place, from the glimpse which I got of it from the train and the little I could walk through the streets. I feared to go very far from the station, as we had arrived late and would start as near the correct time as possible. The impression I had was that we were leaving the West and entering the East; the most western of splendid bridges over the Danube, which is here of noble width and depth, took us among the traditions of Turkish rule.

Calcul de la variance

Définition

La *variance* d'un ensemble de données permet de caractériser la dispersion des valeurs autour de la moyenne. La variance est le carré de l'*écart type*.

La variance V_x d'une population x_i est la moyenne des carrés des écarts des x_i à la moyenne m :

$$V_x = \frac{1}{n} \sum_i (x_i - m)^2$$

Vous pourrez trouver la notation σ^2 pour la variance et μ ou \bar{x} pour la moyenne, selon la discipline (proba ou stats).

Le problème de cette équation, c'est qu'il faut d'abord parcourir les données pour extraire la moyenne.

Autre écriture

Il existe un autre algorithme, en 6 étapes :

1. Nombre de valeurs : $n = \sum_i 1$
2. Somme des valeurs : $S_x = \sum_i x_i$
3. Somme des carrés : $S_{x^2} = \sum_i x_i^2$
4. Moyenne des valeurs : $M_x = \frac{S_x}{n}$
5. Moyenne des carrés : $M_{x^2} = \frac{S_{x^2}}{n}$
6. Variance : $V_x = M_{x^2} - M_x^2$

Cet algorithme ne demande qu'un seul passage dans les données et il est parallélisable avec MapReduce.

Programmation séquentielle

On écrit l'algorithme ainsi :

1. Initialisation :

$$n = Sx = Sx2 = 0$$

2. Pour chaque donnée x :

$$n += 1 ; Sx += x ; Sx2 += x*x$$

3. Terminaison, calculer la variance par :

$$Mx = Sx/n ; Mx2 = Sx2/n ; Vx = Mx2 - Mx*Mx$$

Il reste à transformer cette écriture en MapReduce.

Remarque sur la précision

L'algorithme précédent est peu précis lorsque les nombres sont petits en valeur absolue. Il est alors préférable d'utiliser une **variante** dans laquelle on décale toutes les valeurs d'une même constante.

En effet, $V_{x+K} = V_x$.

On choisit alors $K = x_1$, c'est à dire le premier x des données. Cela donne quelque chose comme ça pour le 2e point page précédente :

```
si n==0 alors K = x
n += 1 ; Sx += (x-K) ; Sx2 += (x-K)*(x-K)
```

mais c'est difficilement applicable dans le cadre MapReduce, car il faut traiter à part l'une des données et transmettre la constante K à toutes les autres, ou alors choisir K arbitrairement.

Écriture MapReduce

Le principe est de ne faire qu'un seul passage à travers les données, lors de la phase *Map*. Cette étape doit collecter les trois informations n , S_x et S_{x^2} . C'est l'étape *Reduce* qui calcule les moyennes et la variance.

- Map
 - extrait la valeur x_i à partir de la donnée courante
 - émet un triplet $(1, x_i, x_i^2)$ en tant que valeur, associé à une clé identique pour tous les triplets
- Combine
 - reçoit une liste de triplets associés à la même clé
 - additionne tous ces triplets pour obtenir (n, S_x, S_{x^2})
- Reduce
 - même calculs que Combine, on obtient les sommes finales
 - calcule M_x , M_{x^2} et V_x

Classe VarianceWritable

Comment transmettre les triplets (n , S_x , S_x^2) entre les trois processus ?

- dériver la classe `ArrayWritable` : pas simple, et le pb, c'est que n est un entier, les autres sont des réels.
- définir notre classe, appelée `VarianceWritable` qui implémente `Writable` :
 - variables membres n , S_x et S_x^2 ,
 - méthodes des `Writable` pour lire et écrire les variables,
 - constructeur : initialise les trois variables à zéro,
 - affectation de x pour faciliter *map*,
 - addition de deux `VarianceWritable` pour faciliter *combine* et *reduce*,
 - calcul des moyennes et de la variance pour faciliter *reduce*.

Classe VarianceWritable (entête)

Voici le début de la classe :



```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Writable;

public class VarianceWritable implements Writable
{
    private long n;
    private double Sx;
    private double Sx2;
```

Remarquez les types : c'est du « Big Data », donc potentiellement, les données peuvent être énormes. Un `int` est limité à ± 2 milliards.

Classe VarianceWritable (constructeur)

Voici le constructeur par défaut de la classe :



```
public VarianceWritable()
{
    clear();
}

public void clear()
{
    n = 0L;
    Sx = 0.0;
    Sx2 = 0.0;
}
```

Le suffixe L indique que c'est une constante long.

Classe VarianceWritable (interface Writable)

Un Writable doit implémenter ces deux méthodes :



```
public void write(DataOutput sortie) throws IOException
{
    sortie.writeLong(n);
    sortie.writeDouble(Sx);
    sortie.writeDouble(Sx2);
}

public void readFields(DataInput entree) throws IOException
{
    n = entree.readLong();
    Sx = entree.readDouble();
    Sx2 = entree.readDouble();
}
```

Il faut lire et écrire exactement la même chose dans le même ordre.

Classe VarianceWritable (méthode pour *map*)


Chaque *mapper* va avoir besoin de produire un triplet initialisé à partir de la valeur courante. La méthode suivante est très utile : ⬇

```
public void set(double x)
{
    n = 1L;
    Sx = x;
    Sx2 = x*x;
}
```

Il suffira de ceci dans le *mapper* pour chaque valeur *x* à traiter :

```
VarianceWritable valeurI = new VarianceWritable();
valeurI.set(x);
context.write(cleI, valeurI);
```

Classe VarianceWritable (méthode pour *combine*)


Les *combiners* et le *reducer* vont devoir additionner de nombreux triplets afin d'obtenir les totaux sur les trois champs. Voici une méthode pratique : 

```
public void add(VarianceWritable autre)
{
    n += autre.n;
    Sx += autre.Sx;
    Sx2 += autre.Sx2;
}
```

Il leur suffira de faire ceci pour chaque liste de valeurs reçues :

```
VarianceWritable valeurS = new VarianceWritable();
for (VarianceWritable valeur: valeursI) {
    valeurS.add(valeur);
}
```

Classe VarianceWritable (méthode pour *reduce*)

Il est pratique de rajouter la méthode de calcul de la variance, plutôt que la coder dans le *reducer*, car toutes les variables sont sur place : 

```
public double getVariance()
{
    double Mx = Sx / n;
    double Mx2 = Sx2 / n;
    return Mx2 - Mx*Mx;
}
```

Avec ça, cette classe peut facilement être intégrée dans un autre projet ou modifiée pour un autre calcul mathématique du même genre (cumul de valeurs).

Classe VarianceWritable (affichage)

Pour finir, on rajoute la méthode d'affichage :



```
public String toString()
{
    return "VarianceWritable(n="+n+", Sx="+Sx+", Sx2="+Sx2+" )";
    // OU return "VarianceWritable("+getVariance()+")";
}
```

Mapper pour la variance

La classe *Mapper* reçoit un texte à traiter. Chaque *thread* va s'occuper d'une seule ligne. Le but est de calculer la variance des longueurs de ligne.

- En entrée du *Mapper*, il y a un texte, donc des paires (LongWritable, Text), parce que le *driver* configure le *job* par :
`job.setInputFormatClass(TextInputFormat.class);`
- En sortie du *Mapper*, on aura des paires (NullWritable, VarianceWritable), la clé étant identique pour toutes les paires.

On a donc l'entête suivant :



```
public class VarianceLongueurLignesMapper
    extends Mapper<LongWritable, Text,
        NullWritable, VarianceWritable>
```

Classe VarianceLongueurLignesMapper

Voici le corps du *mapper*. Les allocations sont faites en dehors. ⬇


```
private NullWritable cleI = NullWritable.get();
private VarianceWritable valeurI = new VarianceWritable();

@Override
public
    void map(LongWritable cleE, Text valeurE, Context context)
        throws IOException, InterruptedException
    {
        valeurI.set( valeurE.getLength() );
        context.write(cleI, valeurI);
    }
```

valeurE contient l'une des lignes du texte à traiter, on place sa longueur dans un *VarianceWritable* en sortie.

Combiner

Ce processus est optionnel de deux manières : d'une part on peut s'en passer, et d'autre part, même programmé, il n'est pas forcément lancé par YARN. S'il est lancé, il est associé à un *Mapper* et il tourne pour réduire les données venant d'une seule machine.

Son rôle est d'avancer le travail du *Reducer*. On va lui demander de calculer les sommes partielles. Comme il se trouve entre le *Mapper* et le *Reducer*, son entête est : 

```
public class VarianceLongueurLignesCombiner
    extends Reducer<NullWritable, VarianceWritable,
                  NullWritable, VarianceWritable>
{
```

On doit remettre les mêmes types en entrée et en sortie pour les clés et les valeurs.

Classe VarianceLongueurLignesCombiner

Voici le corps du *combiner* :



```
private VarianceWritable valeurS = new VarianceWritable();
public void reduce(NullWritable cleI,
                  Iterable<VarianceWritable> valeursI, Context
                  throws IOException, InterruptedException
{
    valeurS.clear();
    for (VarianceWritable valeur : valeursI) {
        valeurS.add(valeur);
    }
    context.write(cleI, valeurS);
}
```

La clé d'entrée est recopiée en sortie et les valeurs d'entrée sont additionnées. On va retrouver ce même schéma dans le *reducer*.

Classe VarianceLongueurLignesReducer

Le *reducer* reçoit toutes les sommes partielles, des *mappers* et éventuellement des *combiners* dans des paires (NullWritable, VarianceWritable) et en sortie, il ne produit qu'un nombre, la variance des longueurs de ligne dans une paire (NullWritable, DoubleWritable).

Voici la définition de la classe :



```
public class VarianceLongueurLignesReducer
    extends Reducer<NullWritable, VarianceWritable,
                   NullWritable, DoubleWritable>
{
```

La clé de sortie sera encore la même clé. C'est comme ça quand on calcule une information synthétique sur la totalité des données. On aurait des clés différentes s'il fallait distinguer différentes variances.

VarianceLongueurLignesReducer (méthode reduce)

Voici le source du *reducer* :



```
private VarianceWritable total = new VarianceWritable();
private DoubleWritable valeurS = new DoubleWritable();

@Override
public void reduce(NullWritable cleI,
                  Iterable<VarianceWritable> valeursI, Context
                  throws IOException, InterruptedException
{
    total.clear(); // indispensable si plusieurs reduce
    for (VarianceWritable valeur : valeursI) {
        total.add(valeur);
    }
    valeurS.set( total.getVariance() );
    context.write(cleI, valeurS);
}
```

Programme principal

Le programme principal crée un job YARN. Il définit les classes, les types des données et les fichiers concernés. Voici son point d'entrée avec la méthode main :



```
public class VarianceLongueurLignesDriver
    extends Configured implements Tool
{
    public static void main(String[] args) throws Exception
    {
        // préparer et lancer un job
        VarianceLongueurLignesDriver driver =
            new VarianceLongueurLignesDriver();
        int exitCode = ToolRunner.run(driver, args);
        System.exit(exitCode);
    }
}
```

Tout est dans la méthode run surchargée de la classe Configured.


Driver

Il a plusieurs choses à faire :

- Vérifier les paramètres. Ils sont dans le tableau `String[] args` passé en paramètre de la méthode.
- Créer le job YARN
- Définir les classes des *Mapper*, *Combiner* et *Reducer* afin que YARN sache quoi lancer
- Définir les types des clés et valeurs sortant du *Mapper*
- Définir les types des clés et valeurs sortant du *Reducer*
- Définir les fichiers ou dossiers à traiter : ce sont les paramètres du programme.
- Lancer le job.

C'est la fonction la plus longue de tout le logiciel.


Classe VarianceLongueurLignesDriver (méthode run)

Voici le début avec la vérification des paramètres et la création du job YARN : 

```
@Override
public int run(String[] args) throws Exception
{
    // vérifier les paramètres
    if (args.length != 2) {
        System.err.println("fournir les dossiers d'entrée et de
        System.exit(-1);
    }

    // créer le job map-reduce
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "VarianceLongueurLignes");
    job.setJarByClass(VarianceLongueurLignesDriver.class);
```

Classe VarianceLongueurLignesDriver (classes)

Voici comment on définit les classes des traitements. On nomme les classes correspondant aux traitements : 


```
// définir les classes Mapper, Combiner et Reducer
job.setMapperClass(VarianceLongueurLignesMapper.class);
job.setCombinerClass(VarianceLongueurLignesCombiner.class);
job.setReducerClass(VarianceLongueurLignesReducer.class);
```

C'est nécessaire car une fois compilé, rien ne dit à YARN quelle classe fait quoi. Le projet suivant montrera une situation où il y a deux MapReduce successifs, donc plusieurs classes pour le même type de travail.

NB: L'archive téléchargeable [VarianceLongueurLignes.tar.gz](#) utilise la classe `YarnJob` qui simplifie énormément ces aspects.

Classe `VarianceLongueurLignesDriver` (entrée)


Voici comment on spécifie les données à traiter. La première instruction dit qu'on va traiter des textes (un fichier CSV, même compressé, est un fichier texte, un .mp3 est un fichier binaire).

La deuxième instruction indique où est le fichier à traiter. Si on fournit un nom de dossier, alors tous les fichiers de ce dossier seront traités. Et avec la troisième ligne, on indique d'aller aussi traiter tous les fichiers des sous-sous-...-dossiers. 

```
// définir les données d'entrée
job.setInputFormatClass(TextInputFormat.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileInputFormat.setInputDirRecursive(job, true);
```

La classe `TextInputFormat` fait que les paires fournies au *mapper* seront des (`LongWritable`, `Text`).

Classe VarianceLongueurLignesDriver (sorties)

Ensuite, on configure les types des clés intermédiaires et finales ainsi que le dossier de sortie : 

```
// sorties du mapper = entrées du reducer et du combiner
job.setMapOutputKeyClass(NullWritable.class);
job.setMapOutputValueClass(VarianceWritable.class);

// définir les données de sortie : dossier et types des pairs
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputKeyClass(NullWritable.class);
job.setOutputValueClass(DoubleWritable.class);
```

Le *reducer* écrira le résultat dans un fichier appelé `part-r-00000` dans le dossier de sortie. Comme c'est une paire (`NullWritable`, `DoubleWritable`), en fait, il n'y aura que le nombre.

Classe VarianceLongueurLignesDriver (lancement)

Pour finir, on lance le job :



```
// lancer le job et attendre sa fin
boolean success = job.waitForCompletion(true);
return success ? 0 : 1;
}
```

La méthode `run` doit retourner un code d'erreur : 0=ok, 1=erreur.
Or la méthode `waitForCompletion` retourne un booléen valant `true` si c'est ok, `false` si ça a planté.

Télécharger le projet complet [VarianceLongueurLignes.tar.gz](#).

Commandes de compilation de cet ensemble

Il reste à voir comment on lance tout ça :

1. **Compilation des sources.** La variable CLASSPATH contient les archives jar nécessaires à la compilation, par exemple
~/lib/hadoop/hadoop-common-2.8.4.jar:
~/lib/hadoop/hadoop-mapreduce-client-core-2.8.4.jar.
Les chemins sont séparés par un :

```
javac -cp $CLASSPATH *.java
```

2. **Création d'une archive jar.** On doit dire quelle est la classe principale, celle qui contient main().

```
jar cfe projet.jar VarianceLongueurLignesDriver *.class
```

C'est plus compliqué si les fichiers source sont placés dans un dossier src et qu'il faut un dossier bin pour mettre les binaires, et aussi s'il y a des paquetages. Voir le Makefile des TP.

Commandes de lancement de cet ensemble

Voici maintenant l'exécution, une fois qu'on a l'archive `projet.jar`, à faire à chaque lancement :

1. **Suppression du dossier des résultats**

```
hdfs dfs -rm -r -f resultats
```

2. **Lancement du job.** On donne les noms des dossiers.

```
yarn jar projet.jar /share/livres resultats
```

3. **Affichage des messages.** Il faut connaître l'identifiant de l'application. Il est affiché dans les premières lignes suivant le lancement. Il est utile de rajouter `| more` à la commande.

```
yarn logs -applicationId application_14579178155_003
```

4. **Affichage des résultats**

```
hdfs dfs -cat resultats/part-r-00000
```


Bilan du projet

- **Écrire le calcul à la manière MapReduce**

L'écriture initiale ne le permet pas. Il faut trouver un moyen de ré-écrire la formule afin de paralléliser les calculs. Ici, ce sont les sommes partielles sur plusieurs données (1 , x et x^2) qui peuvent être faites simultanément par les *combiners*. Par contre, cela demande des connaissances en mathématiques (Théorème de König-Huygens, [wikipedia](#)) qui sont hors compétence.

Si on ne trouve pas les bonnes formules, on ne peut pas faire le travail en MapReduce, ou alors très peu efficacement.

- **Définir la structure de données échangée** entre les *mappers* et le *reducer*

Si on ne peut pas faire avec les types prédéfinis simples, on doit implémenter l'interface `Writable` et rajouter les méthodes utiles. Cela rend souvent le projet plus lisible.

Calcul d'une médiane

Principe

Le programme précédent peut facilement être adapté au calcul de différentes **moyennes** : arithmétique, géométrique (exponentielle de la moyenne des logarithmes), harmonique (inverse de la moyenne des inverses), quadratique (racine de la moyenne des carrés).

La *médiane* est une autre sorte d'information statistique. C'est une valeur qui caractérise un ensemble de données, telle qu'il y a autant de données meilleures que de pires qu'elle.

Par exemple, dans le sac (*bag* ou **multiensemble**) $\{58, 81, 36, 93, 3, 8, 64, 43, 3\}$, la médiane est 43. Il y a autant de valeurs plus petites qu'elle $\{36, 3, 8, 3\}$ que de plus grandes $\{58, 81, 93, 64\}$.

La médiane n'est pas forcément au milieu des données. Par contre, elle l'est quand les données sont triées et en nombre impair.

Principe du calcul en MapReduce

On se propose de calculer la médiane des longueurs des lignes d'un ensemble de textes.

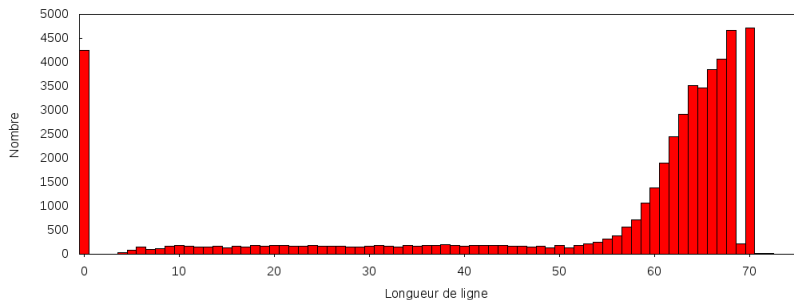
Dans un très grand corpus, de très nombreuses lignes ont la même longueur. Ainsi, ça conduit à un multi-ensemble où le même élément est présent de nombreuses fois. Par exemple, une ligne de longueur 67 est très commune.

Le calcul de la médiane reste le même : il faut trouver la longueur telle qu'il y a autant de lignes plus courtes que de lignes plus longues.

Pour éviter un tri de toutes les longueurs de lignes qui serait très coûteux, le principe est de travailler avec un histogramme des longueurs des lignes. Cet histogramme peut être calculé avec un MapReduce.

Histogramme des longueurs des lignes

Cette image montre le nombre de lignes de chaque longueur. Elle permet de visualiser la position de la médiane, qui est 64 (ici).



La somme des nombres d'un côté, par exemple à gauche entre 0 et 63 est juste inférieure à la moitié du nombre total de lignes.


Données à calculer

En partant des textes organisés en lignes, on va calculer le nombre de lignes de chaque longueur, ainsi que le nombre de lignes total.

- Le nombre de lignes de chaque longueur par un premier MapReduce. Son travail consiste à produire l'histogramme. On va récupérer un tableau (longueur de ligne, nombre de lignes de cette longueur).
- Le nombre de lignes total par un second MapReduce appliqué à l'histogramme (c'est un choix purement pédagogique).

Ensuite, on va calculer la position de la médiane par un parcours de l'histogramme de gauche à droite en accumulant les nombres de lignes. Dès qu'on dépasse la moitié du nombre total, c'est qu'on est sur la médiane.

Premier MapReduce (le *mapper*)

Map1 reçoit des textes ligne par ligne. Chaque thread génère une paire (length(ligne), 1) : 

```
public class MedianeLongueurLignesEtape1Mapper
    extends Mapper<LongWritable, Text, IntWritable, LongWritable>
{
    private final LongWritable valeurI = new LongWritable(1L);
    private IntWritable cleI = new IntWritable();
    @Override
    public void map(LongWritable cleE, Text valeurE, Context cont
        throws IOException, InterruptedException
    {
        cleI.set(valeurE.getLength());
        context.write(cleI, valeurI);
    }
}
```

Premier MapReduce (le *reducer*)

Reduce1 additionne les paires reçues de Map1 :



```
public class MedianeLongueurLignesEtape1Reducer
    extends Reducer<IntWritable, LongWritable,
                    IntWritable, LongWritable> {
    @Override
    public void reduce(IntWritable cleI,
                       Iterable<LongWritable> valeursI, Context context)
        throws IOException, InterruptedException {
        long nombre = 0L;
        for (LongWritable valeurI : valeursI) {
            nombre += valeurI.get();
        }
        LongWritable valeurS = new LongWritable(nombre);
        context.write(cleI, valeurS);
    }
}
```


Premier MapReduce (le *driver*)

Le driver construit un job MapReduce afin de traiter les textes. Il indique un dossier temporaire pour enregistrer les résultats.

Plusieurs MapReduce peuvent être construits et lancés successivement :

```
Configuration conf = this.getConf();

Job etape1 = Job.getInstance(conf, "MedianeLongueurLignes1")
etape1.setJarByClass(MedianeLongueurLignesDriver.class);
...
Job etape2 = Job.getInstance(conf, "MedianeLongueurLignes2")
etape2.setJarByClass(MedianeLongueurLignesDriver.class);
...
if (!etape1.waitForCompletion(true)) return 1;
if (!etape2.waitForCompletion(true)) return 1;
```

Format des fichiers

La particularité, c'est que la sortie du premier MapReduce est mise dans un **SequenceFile** afin d'être facilement relue. C'est un format binaire contenant des paires (clé, valeur).

Voici la configuration de la sortie de l'étape 1 :

```
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
etape1.setOutputFormatClass(SequenceFileOutputFormat.class);
FileOutputFormat.setOutputPath(etape1, PathTMP1);
etape1.setOutputKeyClass(IntWritable.class);
etape1.setOutputValueClass(LongWritable.class);
```

Voici la configuration de l'entrée de l'étape 2 :

```
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
etape2.setInputFormatClass(SequenceFileInputFormat.class);
FileInputFormat.addInputPath(etape2, PathTMP1);
```

Utilisation de la classe YarnJob

Cette classe présentée dans le cours 2 facilite la construction d'un Job MapReduce en vérifiant les types des *Map* et *Reduce*. Son emploi ne change rien pour le premier job, seulement pour le second car le type `SequenceFileInputFormat` n'est pas inspectable :

```
YarnJob etape2 = new YarnJob(conf, "MedianeLongueurLignes2");
etape2.setJarByClass(MedianeLongueurLignesDriver.class);
...

// définir les données d'entrée
etape2.setInputFormatClass(SequenceFileInputFormat.class,
                           IntWritable.class, LongWritable.class);
etape2.addInputPath(PathTMP1);
```

Il faut fournir les types des clés et valeurs du `SequenceFile` d'entrée.

Second MapReduce (le *mapper*)

Map2 reçoit une paire (longueur, nombre) venant du SequenceFile. Il produit une paire (null, nombre) :



```
public class MedianeLongueurLignesEtape2Mapper
    extends Mapper<IntWritable, LongWritable,
                  NullWritable, LongWritable>
{
    // même clé pour tout le monde
    private NullWritable cleI = NullWritable.get();

    @Override
    public void map(IntWritable cleE, LongWritable valeurE, Cont
        throws IOException, InterruptedException
    {
        context.write(cleI, valeurE);
    }
}
```

Second MapReduce (le *reducer*)

En fait, le second *reducer* est identique au premier : il calcule la somme des valeurs par clés. Celui de la seconde étape reçoit des clés `NullWritable`, tandis que le premier reçoit des `IntWritable`.

On peut également utiliser ces *reducer* directement comme *combiner* dans les deux jobs MapReduce.

Le post-traitement


- Le premier MapReduce produit un fichier (temporaire) contenant des paires (longueur de ligne, nombre de lignes de cette longueur). On a vu que c'était un `SequenceFile`.
- Le second MapReduce produit un fichier (temporaire) contenant un seul entier, le nombre de lignes total. C'est un simple fichier texte sur HDFS.

Il y a maintenant deux choses à faire :

1. Récupérer le nombre de lignes total issu du second MapReduce
2. Récupérer et traiter l'histogramme issu du premier MapReduce


Ce qui est important, c'est que YARN fait trier les lignes sortant du *reducer* dans l'ordre croissant des clés. C'est à dire l'histogramme sera automatiquement dans l'ordre.

Récupérer le nombre total de lignes

Il faut relire le fichier part-r-00000 issu du second MapReduce. Il ne contient qu'un entier à lire : 

```
private long ReadTotal() throws IOException
{
    FSDataInputStream inStream =
        fs.open(new Path(PathTMP2, "part-r-00000"));
    try {
        InputStreamReader isr = new InputStreamReader(inStream);
        BufferedReader br = new BufferedReader(isr);
        String line = br.readLine();
        return Integer.parseInt(line);
    } finally {
        inStream.close();
    }
}
```

Parcourir l'histogramme

Il faut relire le SequenceFile issu du premier MapReduce. C'est un peu plus complexe : 

```
SequenceFile.Reader.Option fich =
    SequenceFile.Reader.file(new Path(PathTMP1,"part-r-00000"));
SequenceFile.Reader reader = new SequenceFile.Reader(conf, fich)
try {
    IntWritable longueur = new IntWritable();
    LongWritable nombre = new LongWritable();
    while (reader.next(longueur, nombre)) {
        // traiter le couple (longueur, nombre)
        ...
    }
    System.out.println("resultat : "+resultat);
} finally {
    reader.close();
}
```


Calculer la médiane

C'est de l'algorithmique standard. Dans la boucle précédente, on parcourt les couples (longueur, nombre) dans l'ordre croissant des longueurs. Il faut juste arrêter la boucle quand la somme des nombres vus jusque là dépasse la moitié du nombre total.

```
// moitié du total
long limite = ReadTotal() / 2;
long cumul = 0L;
// tant qu'on n'atteint pas la moitié du total
while (reader.next(longueur, nombre) && cumul < limite) {
    cumul += nombre.get();
}
return longueur.get();
```

On peut améliorer pour retourner une interpolation entre la longueur actuelle et sa précédente, selon le dépassement de la limite.

Bilan du projet

Cet exemple a montré comment exploiter les résultats d'un ou plusieurs MapReduce dans un même programme.

Plusieurs formats de fichiers peuvent être employés : fichiers textes CSV et fichiers de paires (clés, valeurs).

Télécharger le projet complet MedianeLongueurLignes.tar.gz.