BigData - Semaine 2

Pierre Nerzic

février-mars 2019

Le cours de cette semaine présente davantage de détails sur les jobs MapReduce dans YARN :

- spécification des entrées
- spécification des paires (clé, valeurs)
- spécification des sorties
- traitement de certains fichiers
- MapReduce dans d'autres langages sur YARN

Pierre Nerzic

Jobs MapReduce

Pierre Nerzic

Création et lancement d'un Job

Revenons sur le lancement d'un job MapReduce : 

```
public int run(String[] args) throws Exception
{
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "traitement");
    job.setJarByClass(TraitementDriver.class);
    job.setMapperClass(TraitementMapper.class);
    job.setReducerClass(TraitementReducer.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
}
```


Pierre Nerzic

Spécification des entrées

Les lignes suivantes spécifient ce qu'on veut traiter : 

```
FileInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);
```

- La première ligne indique quels sont les fichiers HDFS à traiter,
- La seconde ligne indique le type de contenu de ces fichiers.

Voici davantage d'informations sur ces instructions.

Fichiers d'entrée

Cette instruction indique où prendre les fichiers à traiter : 

```
FileInputFormat.addInputPath(job, new Path("NOMCOMPLET"));
```

C'est un appel à une méthode statique dans la classe `FileInputFormat`.

- Si le chemin fourni est un dossier, alors tous ses fichiers sont employés,
- Si les fichiers trouvés sont compressés (extensions `.gz`, `.bz2`, `.lzo`...), ils sont automatiquement décompressés.

Les sous-classes de `FileInputFormat` telles que `TextInputFormat` et `KeyValueTextInputFormat` sont également responsables de la séparation (*split*) des données en paires (clé,valeur).

Format des données d'entrée

Cette instruction spécifie le type des fichiers à lire et implicitement, les clés et les valeurs rencontrées :

```
job.setInputFormatClass(TextInputFormat.class);
```

Important: les types des clés et valeurs du `Mapper` doivent coïncider avec la classe indiquée pour le fichier.

Ici, la classe `TextInputFormat` est une sous-classe de `FileInputFormat<LongWritable,Text>`. Donc il faut écrire :

```
public class TraitementMapper
        extends Mapper<LongWritable,Text, TypCleI,TypValI>
{
    @Override
    public void map(LongWritable cleE, Text valE, ...
```

Autres formats d'entrée

Il existe d'autres formats d'entrée, comme `KeyValueTextInputFormat` qui est capable de lire des fichiers déjà au format (clé, valeur) :

- les lignes se finissent par un `'\n'` ou un `'\r'` (cause sûrement un pb avec des fichiers Windows qui ont les deux à la fois)
- chaque ligne est un couple (clé, valeur)
- c'est une tabulation `'\t'` qui sépare la clé de la valeur
- ces deux informations sont des `Text`

```
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

```
public class TraitementMapper
        extends Mapper<Text,Text, TypCleI,TypValI>
```

Changement du séparateur de KeyValueTextInputFormat

On peut changer le séparateur, par exemple une virgule :

```
Configuration conf = new Configuration();
conf.set(
    "mapreduce.input.keyvaluelinerecordreader          \
                                    .key.value.separator",
    ",");

Job job = Job.getInstance(conf, "nom");
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

NB: dans le premier paramètre de `conf.set`, enlever le `\` et mettre tout sur la même ligne.

Format des données intermédiaires

Les types des clés et valeurs sortant du *mapper* et allant au *reducer*, notés `TypCleI` et `TypValI` dans ce qui précède, sont définis par les instructions suivantes : 

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
```

Elles forcent la définition du *mapper* et du *reducer* ainsi :

```
class TraitementMapper extends Mapper<..., Text, IntWritable>
class TraitementReducer extends Reducer<Text, IntWritable, ...>
```

Elles sont absolument obligatoires quand ce ne sont pas les types par défaut, `ClassCastException` lors du lancement du *reducer* sinon.

Format des données de sortie

Voici les instructions qui spécifient le format du fichier de sortie : 📥

```
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
```

Ce doivent être les types de sortie du *Reducer* :

```
class TraitementReducer
        extends Reducer<..., Text, DoubleWritable>
```

La classe `TextOutputFormat<K,V>` est paramétrée par les types des clés et des valeurs. Par défaut, ce sont tous deux des `Text`.

Il existe d'autres classes pour produire les données de sortie (voir plus loin), dont des écrivains sur mesure (voir en TP).

Fichiers de sortie

Les résultats du job sont enregistrés dans des fichiers situés dans le dossier indiqué par :

```
FileOutputFormat.setOutputPath(job, new Path("DOSSIER"));
```

YARN enregistre un fichier par Reducteur final. Leurs noms sont `part-r-00000`, `part-r-00001`,...

La classe `Path` possède plusieurs constructeurs qui permettent de concaténer des chemins :

```
Path sortie1 = new Path("/tmp", "MonAppli", "etape1");
```

Définit `sortie1` valant `/tmp/MonAppli/etape1`

Post-traitement des résultats

Au lieu de récupérer un simple fichier, on peut afficher proprement le résultat final :

```
job.setOutputFormatClass(SequenceFileOutputFormat.class);
if (job.waitForCompletion(true)) {
   SequenceFile.Reader.Option fichier =
      SequenceFile.Reader.file(new Path(sortie,"part-r-00000"));
   SequenceFile.Reader reader =
      new SequenceFile.Reader(conf, fichier);
   IntWritable annee = new IntWritable();
   FloatWritable temperature = new FloatWritable();
   while (reader.next(annee, temperature)) {
      System.out.println(annee + " : " + temperature);
   }
   reader.close();
}
```

Types des clés et valeurs

Pierre Nerzic

Type `Writable`

Nous avons vu la semaine dernière qu'il fallait employer des `Writable` : `Text`, `IntWritable`, `FloatWritable`. L'interface `Writable` est une optimisation/simplification de l'interface `Serializable` de Java. Celle de Java construit des structures plus lourdes que celle de Hadoop, parce qu'elle contiennent les noms des types des données, tandis que les `Writable` ne contiennent que les octets des données, et d'autre part les `Writable` sont modifiables (*mutables*).

Il existe différents types de `Writable` pour des collections. On va donner l'exemple d'un `Writable` spécifique dérivant d'un type tableau.

Classe ArrayWritable

Le type ArrayWritable représente des tableaux de Writable quelconques. Il est préférable de la sous-classer pour qu'elle contienne les données voulues :

```
public class IntArrayWritable extends ArrayWritable {
   public IntArrayWritable() { super(IntWritable.class); }
   public IntArrayWritable(int size) {
      super(IntWritable.class);
      IntWritable[] values = new IntWritable[size];
      for (int i=0; i<size; i++) values[i] = new IntWritable();
      set(values);
   }
   public IntWritable itemAt(int index) {
      Writable[] values = get();
      return (IntWritable)values[index];
   }
}
```

Emploi de cette classe

Voici comment créer et utiliser une telle structure :

```
public class TraitementMapper
    extends Mapper<LongWritable, Text, Text, IntArrayWritable>
{
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException
  {
     Text cle = new Text("ok");
     IntArrayWritable valeur = new IntArrayWritable(2);
     valeur.itemAt(0).set(123);
     valeur.itemAt(1).set(value.getLength());
     context.write(cle, valeur);
  }
}
```

Méthodes supplémentaires

Vous pouvez rajouter vos propres méthodes à cette classe. Par exemple pour additionner un autre `IntArrayWritable` à `this` :

```
public void add(IntArrayWritable autre)
{
    // récupérer les valeurs
    Writable[] values = this.get();
    Writable[] autres = autre.get();

    // this = this + autre
    for (int i=0; i<values.length; i++) {
        IntWritable val = (IntWritable)values[i];
        IntWritable aut = (IntWritable)autres[i];
        val.set(val.get() + aut.get());
    }
}
```

Remarques

En pratique, on n'utilisera pas cette technique car c'est assez peu maniable, les méthodes sont compliquées et on peut se tromper d'indices pour les cases du tableau.

Au contraire, on préfèrera employer la technique des transparents suivants, consistant à implémenter l'interface `Writable` dans une classe spécifique. C'est assez simple à faire et ça rend les programmes extrêmement faciles à comprendre.

Interface `Writable`

L'interface `Writable` gère des contenus transmis entre un *mapper* et un *reducer*. Pour l'implémenter, il suffit de deux méthodes :

- `public void write(DataOutput sortie)` : elle écrit des données sur `sortie`,
- `public void readFields(DataInput entree)` : vous devez extraire les mêmes données et dans le même ordre.

Les deux classes `DataInput` et `DataOutput` sont des sortes de flots binaires (*binary stream*), comme des fichiers. Ils possèdent toutes les méthodes pour lire/écrire les types courants :

- `DataInput` : `readBoolean()`, `readInt()`, `readFloat()`, `readLine()`, etc.
- `DataOutput` : `writeBoolean(b)`, `writeInt(i)`, `writeFloat(f)`, `writeLine(s)`...

Exemple d'un `Writable`

Voici un exemple pour calculer une moyenne dans le *reducer* :

```
public class MoyenneWritable implements Writable
{
    private double total = 0.0;
    private long nombre = 0L;

    public void write(DataOutput sortie) throws IOException {
        sortie.writeDouble(total);
        sortie.writeLong(nombre);
    }

    public void readFields(DataInput entree) throws IOException
        total  = entree.readDouble();
        nombre = entree.readLong();
    }
}
```


Pierre Nerzic

Constructeurs

Il est nécessaire d'ajouter le constructeur par défaut, sans paramètre.
D'autres constructeurs peuvent être ajoutés : 

```
public MoyenneWritable() {
    total = 0.0;
    nombre = 0L;
}

public MoyenneWritable(double valeur) {
    total = valeur;
    nombre = 1L;
}
```

Le constructeur par défaut est utilisé de manière implicite lors de la désérialisation. Si on ne le met pas, cela plante le job MapReduce.

Méthodes supplémentaires

On peut lui ajouter des méthodes pour faciliter la programmation du *mapper* et du *reducer* :

```
public void set(double valeur) {
    total = valeur;
    nombre = 1L;
}

public void add(MoyenneWritable autre) {
    total  += autre.total;
    nombre += autre.nombre;
}

public double getMoyenne() {
    return total/nombre;
}
```


Pierre Nerzic

Méthodes diverses

On peut rajouter une méthode d'affichage `toString()`. Elle est utilisée implicitement en sortie du *Reducer* :

```
    public String toString() {
        return "Moyenne(total="+total+", nombre="+nombre+")";
        // OU return "MoyenneWritable("+getMoyenne()+")";
    }
```

Utilisation dans un *Mapper*

Voici comment on peut l'employer côté *mapper* :

```
public class MoyenneHauteurArbresMapper
    extends Mapper<LongWritable, Text, Text, MoyenneWritable>
{
   @Override
   public void map(LongWritable cleE, Text valeurE, Context cont
   {
      Arbre.fromLine(valeurE.toString());
      Text cleI = new Text(Arbre.getGenre());
      MoyenneWritable valeurI = new MoyenneWritable();
      valeurI.set(Arbre.getHauteur());
      context.write(cleI, valeurI);
   }
}
```

NB: il manque tout ce qui est exception et filtrage des lignes.

Utilisation dans un *Reducer*

Et côté *reducer* :

```
public class MoyenneHauteurArbresReducer
    extends Reducer<Text, MoyenneWritable, Text, DoubleWritable>
{
   @Override
   public void reduce(Text cleI,
            Iterable<MoyenneWritable> valeursI, Context context)
   {
      // cumuler les valeursI
      MoyenneWritable moyenne = new MoyenneWritable();
      for (MoyenneWritable moy : valeursI) {
         moyenne.add(moy);
      }
      valeurS = new DoubleWritable(moyenne.getMoyenne());
      context.write(cleI, valeurS);
   }
}
```


Pierre Nerzic

Configuration du *Driver*

Pour finir, voici le cœur du *driver* :

```
Configuration conf = this.getConf();
Job job = Job.getInstance(conf, "MoyenneHauteurArbres Job");
job.setJarByClass(MoyenneHauteurArbresDriver.class);
job.setMapperClass(MoyenneHauteurArbresMapper.class);
job.setReducerClass(MoyenneHauteurArbresReducer.class);
FileInputFormat.addInputPath(job, new Path("arbres.csv"));
job.setInputFormatClass(TextInputFormat.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(MoyenneWritable.class);
FileOutputFormat.setOutputPath(job, new Path("resultats"));
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
boolean success = job.waitForCompletion(true);
```

Classe NullWritable

Quand les clés ou valeurs sont sans importance, au lieu de mettre une valeur quelconque, on peut employer le type `NullWritable` :

```
public class MonMapper
        extends Mapper<..., NullWritable, IntWritable>
{
    private NullWritable cleI = NullWritable.get();
    private IntWritable valeurI = new IntWritable();

    @Override
    public void map(..., Context context) throws ...
    {
        ...
        // émettre une paire (clé, valeur)
        context.write(cleI, valeurI);
    }
}
```


Pierre Nerzic

Jobs en fonctionnement

Pierre Nerzic

Contrôle des types des classes

Un job MapReduce est composé de plusieurs classes qui doivent être appariées :

- Les entrées de la classe Map doivent correspondre au format des fichiers d'entrée. Par exemple `job.setInputFormatClass(TextInputFormat.class);` impose des clés `LongWritable` et des valeurs `Text`.
- S'il y a un combiner, voir plus loin, les sorties de la classe Map doivent correspondre aux entrées de la classe Combiner et les sorties du Combiner doivent correspondre à la fois à ses propres entrées et à celles du Reducer.
- Dans tous les cas, les sorties de la classe Map doivent correspondre aux entrées de la classe Reducer.
- Le format du fichier de sortie s'adapte aux sorties de la classe Reducer.

Pierre Nerzic

Exemple d'erreur

Ce job, configuré ainsi, est incorrect pour plusieurs raisons :

```
job.setInputFormatClass(TextInputFormat.class);
```

avec

```
public class MonMapper extends Mapper
    <IntWritable, Text, IntWritable, DoubleWritable>
```

```
public class MonReducer extends Reducer
    <Text, FloatWritable, Text, DoubleWritable>
```

Mais cela ne se verra qu'à l'exécution seulement : `ClassCastException` à chaque appel à `map` et à `reduce`. Le compilateur Java ne peut pas détecter ces erreurs, car elles sont dans des classes différentes, qui ne s'appellent pas directement.

Détecter les erreurs

Pour éviter ça, une classe YarnJob sera proposée en TP : 

```
Configuration conf = this.getConf();
YarnJob job = new YarnJob(conf, "MonMapReduce Job");
job.setJarByClass(TraitementDriver.class);
job.setMapperClass(TraitementMapper.class);
job.setReducerClass(TraitementReducer.class);
job.setInputFormatClass(TextInputFormat.class);
job.addInputPath(new Path(args[0]));
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputPath(new Path(args[1]));
boolean success = job.waitForCompletion(true);
```

Elle vérifie les types à l'aide de l'introspection Java, et empêche toute exécution en cas de non-correspondance. Vous aurez peut-être à la modifier dans le cas d'un projet non standard.

Remarque importante sur l'efficacité

Il faut éviter toute allocation mémoire répétée comme :

```
for (int i=0; i<10000; i++) {
    IntWritable valeur = new IntWritable(i);
    ...
}
```

Il vaut mieux créer les objets hors de la boucle et utiliser leur modificateur ainsi :

```
IntWritable valeur = new IntWritable();
for (int i=0; i<10000; i++) {
    valeur.set(i);
    ...
}
```

C'est possible parce que les `Writable` sont réaffectables (*mutables*).

Allocation en dehors des méthodes

En poursuivant de la même manière, on enlève les allocations des méthodes :

```
public class TraitementMapper extends Mapper<...>
{
    private TypCleI cleI = new TypCleI(...);
    private TypValI valI = new TypValI(...);
    @Override
    public void map(TypCleE cleE, TypValE valE, Context context)
    {
        cleI.set(...);
        valI.set(...);
        context.write(cleI, valI);
    }
}
```


Pierre Nerzic

Piège à éviter

N'oubliez pas que le *reducer* peut être relancé plusieurs fois :

```
public class TraitementReducer extends Reducer<...>
{
    private TypValI valS = new TypValS(...);
    private Moyenne moyenne = new Moyenne();
    @Override
    public void reduce(Text cleI, Iterable<Moyenne> valeursI, ..
    {
        // BUG : moyenne n'a pas été remise à zéro !!!
        for (Moyenne valeurI : valeursI) {
            moyenne.add(valeurI);
        }
        valeurS = new DoubleWritable(moyenne.getMoyenne());
        context.write(cleI, valeurS);
    }
}
```


Pierre Nerzic

Entre Map et Reduce

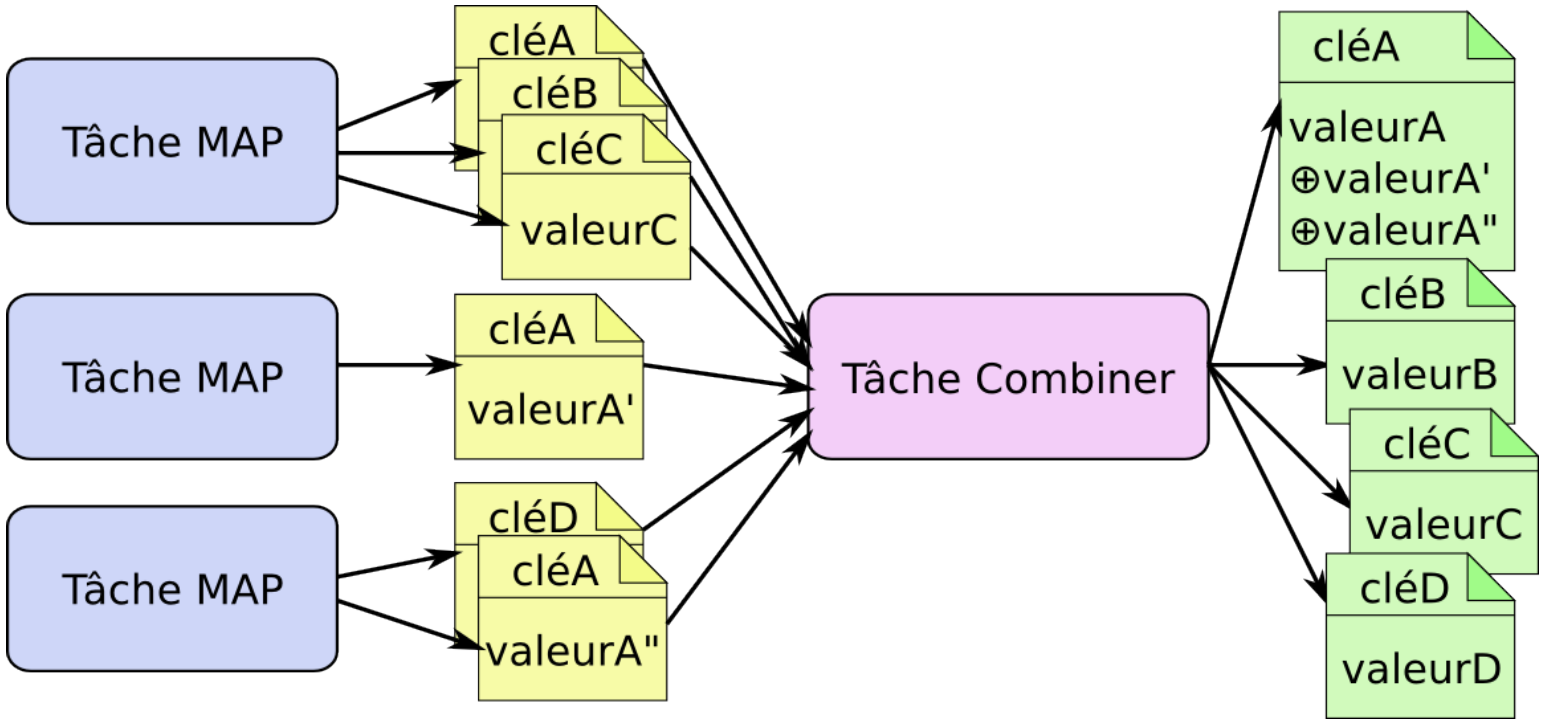
Pierre Nerzic

Combiner

Pour l'instant, nous avons vu deux sortes de tâches : *map* et *reduce*. Nous avons vu que les paires (clé,valeur) produites par les tâches *map* sont envoyées par une étape appelée *shuffle and sort* à travers le réseau vers les tâches *réduce* de manière à regrouper toutes les clés identiques sur la même machine.

Quand on traite des données volumineuses, ça peut devenir trop lent. Hadoop propose un troisième intervenant, entre *map* et *reduce* qui effectue un traitement local des paires produites par *map*. C'est le « *Combiner* ». Son travail est de faire une première étape de réduction de tout ce qui est produit sur une machine.

Schéma du *Combiner*



Il traite des paires ayant la même clé sur la même machine que les tâches *Map*. Les paires qu'il émet sont envoyées aux *reducers*.

Cas d'emploi d'un *Combiner*

Le *combiner* permet de gagner du temps, non seulement en regroupant les valeurs présentes sur la même machine, mais en faisant un petit calcul au passage.

Par exemple, pour calculer la plus grande valeur d'un ensemble de données, il peut calculer le maximum de ce qu'il reçoit localement, et au lieu de produire une liste de valeurs à destination des réducteurs, il n'en produit qu'une seule.

De fait, dans certains cas, le *combiner* est identique au *reducer*. On peut utiliser la même classe pour les deux.

```
job.setMapperClass(TraitementMapper.class);
job.setCombinerClass(TraitementReducer.class);
job.setReducerClass(TraitementReducer.class);
```

Cas de non-emploi d'un *Combiner*

Que pensez-vous de ceci ? On veut calculer la température moyenne par station météo depuis 1901. Les tâches *map* parcourent les relevés, extraient l'identifiant de la station météo et la température relevée, ceci pour chaque mesure. Les paires sont (idstation, temperature). Les tâches *reduce* calculent la moyenne par station météo.

Peut-on utiliser un *combiner* chargé de calculer la moyenne des température par station sur chaque machine contenant un *map* ?

On ne peut pas employer de *combiner* quand l'opérateur d'agrégation n'est pas commutatif ou pas associatif. Les opérateurs *somme*, *min* et *max* sont commutatifs et associatifs, mais pas le calcul d'une moyenne.

Différences entre un *Combiner* et un *Reducer*

1. Les paramètres d'entrée et de sortie du *Combiner* doivent être identiques à ceux de sortie du *Mapper*, tandis que les types des paramètres de sortie du *Reducer* peuvent être différents de ceux de son entrée.
2. On ne peut pas employer un *Combiner* quand la fonction n'est pas commutative et associative.
3. Les *Combiners* reçoivent leurs paires d'un seul *Mapper*, tandis que les *Reducers* reçoivent les paires de tous les *Combiners* et/ou tous les *Mappers*. Les *Combiners* ont une vue restreinte des données.
4. Hadoop n'est pas du tout obligé de lancer un *Combiner*, c'est seulement une optimisation locale. Il ne faut donc pas concevoir un algorithme « map-combine-reduce » dans lequel le *Combiner* jouerait un rôle spécifique.

Squelette de *Combiner*

Les *combiners* reprennent la même interface que les *reducers* sauf que les paires de sortie doivent être du même type que les paires d'entrée :

```
public class TraitementCombiner
        extends Reducer<TypCleI,TypValI, TypCleI,TypValI>
{
    @Override
    public void reduce(TypCleI cleI, Iterable<TypValI> listeI,
                Context context) throws Exception
    {
        for (TypValI val: listeI) {
        /** traitement: cleS = ..., valS = ... **/
        }
        context.write(new TypCleI(cleS), new TypValI(valS));
    }
}
```

MapReduce dans d'autres langages

Présentation

Hadoop permet de programmer un Job MapReduce dans d'autres langages que Java : Ruby, Python, C++... En fait, il suffit que le langage permette de lire `stdin` et écrive ses résultats sur `stdout`.

- Le *Mapper* est un programme qui lit des lignes sur stdin, calcule ce qu'il veut, puis écrit des lignes au format `"%s\t%s\n"` (clé, valeur) sur la sortie.
- Entretemps, les lignes sont triées selon la clé, exactement comme le ferait la commande Unix `sort`
- Le *Reducer* doit lire des lignes au format `"%s\t%s\n"` (clé, valeur). Il doit d'abord séparer ces deux informations, puis traiter les lignes successives ayant la même clé. Ça vous rappellera la commande Unix `uniq -c`.

Exemple de *Mapper* en Python

Voici le « compteur de mots » programmé en Python : 

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
# traiter chaque ligne de l'entrée standard
for ligne in sys.stdin:
    # couper en mots et traiter chacun d'eux
    for mot in ligne.split():
        # trivialement: ce mot est en 1 exemplaire
        paire = (mot, 1)
        # écrire la paire
        print '%s\t%s' % paire
```

Chaque ligne est découpée en mots ; chacun est écrit en tant que clé, avec la valeur 1 sur stdout.

Algorithme du réducteur

C'est un peu plus compliqué. C'est de l'algorithmique. Vous allez recevoir N lignes composées de paires (clé,valeur). Vous devez accumuler (somme, moyenne, min, max...) les valeurs correspondant à des clés identiques. Vous savez que les clés sont triées dans l'ordre, donc des lignes successives auront la même clé, sauf quand on change de clé.

Le parcours se fait ligne par ligne. Il faut donc mémoriser la clé de la ligne précédente ainsi que l'accumulation des valeurs de cette clé. Quand la ligne courante a la même clé que la précédente, on met à jour le cumul. Sinon, on affiche le cumul (et sa clé) sur la sortie et on le remet à zéro.

À la fin, ne pas oublier d'afficher la dernière clé et le cumul.

Exemple de *Reducer* en Python

Voici le cœur du compteur de mots sans les premières lignes : 

```
cle_prec,nombre_total = None,0
for ligne in sys.stdin:
    cle, valeur = ligne.split('\t', 1)
    if cle == cle_prec:
        nombre_total += int(valeur)
    else:
        if cle_prec != None:
            paire = (cle_prec, nombre_total)
            print '%s\t%s' % paire
        cle_prec = cle
        nombre_total = int(valeur)
if cle_prec != None:
    paire = (cle_prec, nombre_total)
    print '%s\t%s' % paire
```


Lancement de ce Job

Pour lancer un tel job, il faut

- placer les deux scripts `mapper.py` et `reducer.py` sur HDFS
- taper la commande complexe suivante :

```
yarn jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
    -files mapper.py,reducer.py \
    -mapper mapper.py  -reducer reducer.py \
    -input livres  -output sortie
```

Il est souhaitable d'en faire un script ou un `Makefile`.

Il est intéressant de voir qu'on peut le lancer dans un tube Unix, mais en mode séquentiel :

```
cat data | mapper.py | sort | reducer.py
```