

DATUM ACADEMY



Hadoop, Spark & Map/Reduce

Benjamin Renaut
Mis à jour par Sergio Simonian

Plan

Module 1 :

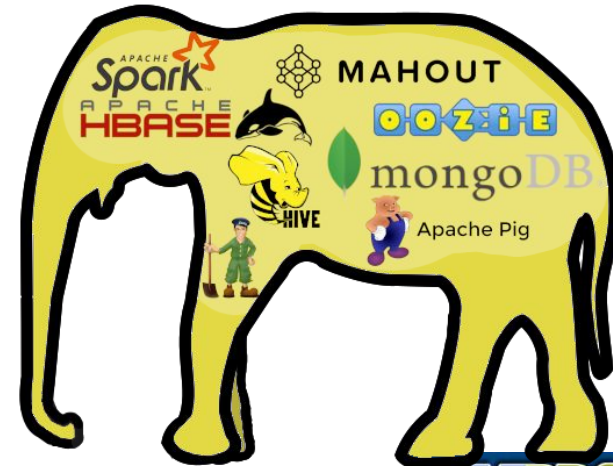
- Big Data et le calcul distribué
- Le paradigme Map/Reduce
- Introduction à Hadoop

Module 2 :

- Programmation Hadoop

Module 3 :

- Spark
- Écosystème autour d'Hadoop



Programmation Hadoop avancé

Propriétés de configuration, Compteurs,
Classe InputFormat, Classe OutputFormat, Interface Writable

Passage d'informations à un Mapper/Reducer

- Reprenons l'exemple de comptage des mots : nous voulons y ajouter une nouvelle fonctionnalité optionnelle - mettre les mots en majuscules.
- Dans ce cas une manière de procéder serait :
 - ◆ Créer 2 classes "Mapper" - une pour les lettres minuscules et une pour les majuscules.

Passage d'informations à un Mapper/Reducer

Solution : 2 classes "Mapper" - dans le main de la classe driver :

```
String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
Job job = Job.getInstance(conf, "WordCount v1.0");
// ...
if(ourArgs[0].equals("--uppercase"))
    job.setMapperClass(WordCountMapUpperCase.class);
else
    job.setMapperClass(WordCountMap.class);
// ...
FileInputFormat.addInputPath(job, new Path(ourArgs[1]));
FileOutputFormat.setOutputPath(job, new Path(ourArgs[2]));
```

Passage d'informations à un Mapper/Reducer

Solution : 2 classes "Mapper" :

- Pas idéal : la plupart du code du Mapper est répété.
- Très limité : que faire si le paramètre optionnel n'est pas booléen ?

Solution : "propriétés de configuration" :

- Les propriétés de configuration nous permettent de définir des valeurs dans la classe Driver et de les récupérer dans les classes Mapper/Reducer.
- Ainsi notre classe Mapper serait consciente si nous voulions mettre les mots en majuscule ou pas.

Passage d'informations à un Mapper/Reducer

→ Pour définir une valeur :

```
conf.setInt("org.ebihar.hadoop.wordcount.uppercase", 1);
```

→ Il y a de nombreux fonctions pour définir d'autres types de données :

◆ setLong, setFloat, etc. et set pour le type String.

→ **Pour récupérer une valeur** : L'objet de configuration est récupéré avec l'objet **Context** dans les fonctions map/reduce.

```
Configuration conf = context.getConfiguration();  
conf.getInt("org.ebihar.hadoop.wordcount.uppercase", 0)
```

→ Il y a de nombreuses autres fonctions: getInt, getLong, etc. et get pour le type String.

→ Le second argument spécifie **une valeur par défaut** si la propriété n'est pas trouvée.

Passage d'informations à un Mapper/Reducer

Solution : propriétés de configuration - dans le main de la classe driver :

```
String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  
// ...  
int uppercase = 0;  
if(ourArgs[0].equals("--uppercase"))  
    uppercase = 1;  
conf.setInt("org.embds.hadoop.wordcount.uppercase", uppercase);  
Job job = Job.getInstance(conf, "WordCount v1.0");  
// ...  
FileInputFormat.addInputPath(job, new Path(ourArgs[1]));  
FileOutputFormat.setOutputPath(job, new Path(ourArgs[2]));
```

Passage d'informations à un Mapper/Reducer

Solution : propriétés de configuration - dans la classe Mapper :

```
protected void map(Object offset, Text value, Context context)
throws IOException, InterruptedException {
    StringTokenizer tok;
    Configuration conf = context.getConfiguration();
    int uppercase = conf.getInt("org.embds.hadoop.wordcount.uppercase", 0);
    if(uppercase != 0)
        tok = new StringTokenizer(value.toString().toUpperCase(), " ");
    else
        tok = new StringTokenizer(value.toString(), " ");
    while(tok.hasMoreTokens()) {
        Text word = new Text(tok.nextToken());
        context.write(word, new IntWritable(1));
    }
}
```

Compteurs Hadoop

- Reprenons l'exemple du parcours de graph.
- Notre **condition d'arrêt était**: l'état couleur de tous les nœuds est **NOIRE**.
- Une façon d'implémenter cette condition serait :
 - ◆ Après chaque exécution - Effectuer une recherche dans l'ensemble des données de sortie pour les nœuds qui ne sont pas de couleur noire.

Compteurs Hadoop

Exemple:

```
private static Boolean conditionDArret(Configuration conf, String output_path) {  
    FileSystem fs = FileSystem.get(conf);  
    FileStatus[] list = fs.globStatus(new Path(output_path));  
    for(int i = 0; i < list.length; ++i) {  
        BufferedReader br = new BufferedReader(new  
InputStreamReader(fs.open(list[i].getPath())));  
        String line = br.readLine();  
        while(line != null) {  
            if(!line.contains("NOIR")) {  
                br.close();  
                return(false);  
            }  
            line=br.readLine();  
        }  
        br.close();  
    }  
    return(true);  
}
```

Compteurs Hadoop

Cette solution est problématique :

- Le graphe pourrait contenir des millions de nœuds.
- Notre programme lit la sortie 2 fois. (Dans **reduce** et **conditionDArret**)

Une meilleure solution serait :

- Vérifier la condition d'arrêt pendant l'exécution de **reduce**.
- Transmettre cette information au Driver.
Pour cela les compteurs Hadoop nous seront utiles.

Compteurs Hadoop

Un compteur Hadoop :

- Contient une valeur numérique : de type Long.
- Ne peut être qu'incrémenté.
- Dans Mapper/Reducer : peut être lu et incrémenté.
- Dans Driver : peut être lu.

Cas d'utilisation :

- Communication entre Driver et les Mappers/Reducers.
- Collecter des statistiques sur un Job en cours.
- Détection des problèmes (Par exemple : compter le nombre d'erreurs)

Compteurs Hadoop

- Pour créer un compteur, déclarez une énumération. (par exemple dans le Driver) :

```
public enum GRAPH_COUNTERS {  
    NB_NODES_UNFINISHED // Amount of non-black nodes after one execution cycle  
};
```

- Le compteur peut ensuite être récupéré soit à partir de l'objet Job (dans le Driver) ou Contexte (dans Mapper/Reducer):

```
Counters cn = job.getCounters();  
Counter cnt = cn.findCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);
```

```
Counter cnt = context.getCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);
```

Compteurs Hadoop

→ Pour **lire** la valeur d'un compteur (Dans Driver/Mapper/Reducer) :

```
Counter cnt;  
// ...  
Long value = cnt.getValue();
```

→ Pour l'**incrémenter** (uniquement dans Mapper/Reducer) :

```
Counter cnt;  
// ...  
cnt.increment(1);
```

→ Note: Peut également être incrémenté de plus de 1.

Compteurs Hadoop

→ Notre fonction de condition d'arrêt dans Driver devient :

```
private static Boolean conditionDArret(Job previous_job) {  
    Counters cn = previous_job.getCounters();  
    Counter cnt = cn.findCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);  
    // Si le compteur n'a jamais été incrémenté : nous n'avons pas rencontré  
    // un nœud qui n'était pas de couleur noire dans les Reducers  
    // de l'exécution du Job spécifié.  
    return(cnt.getValue() == 0);  
}
```

Compteurs Hadoop

→ Et à la fin de notre Reducer, ajoutons quelque chose comme :

```
if(!new_node_color.equals("NOIR"))  
    context.getCounter(Graph.GRAPH_COUNTERS.NB_NODES_UNFINISHED).increment(1);
```

→ Ainsi le Reducer communique la condition d'arrêt au Driver et nous évitons la relecture de la sortie.

Compteurs Hadoop

Hadoop possède également des compteurs internes; la plupart d'entre eux se trouvent dans TaskCounter et JobCounter (package org.apache.hadoop.mapreduce).

Quelques exemples :

- **TOTAL_LAUNCHED_MAPS** : Le nombre de tâches MAP qui ont été lancées. Inclut les tâches qui ont été lancées de manière spéculative.
- **TOTAL_LAUNCHED_REDUCEs** : Le nombre de tâches REDUCE qui ont été lancées. Inclut les tâches qui ont été lancées de manière spéculative.
- **NUM_FAILED_MAPS** : Le nombre de tâches MAP qui ont échoué.
- **NUM_FAILED_REDUCEs** : Le nombre de tâches REDUCE qui ont échoué.
- **REDUCE_INPUT_GROUPS** : Le nombre de groupes clef-valeur distincts envoyés aux Reducers.

Classe InputFormat

- Jusqu'à présent : l'entrée était toujours un fichier texte sur HDFS, divisé par ligne.
- Produit des couples clef/valeur : la clef passée au Mapper est **le numéro de ligne**, la valeur est la **ligne elle-même**.
- Comportement par défaut d'Hadoop - **peut être modifié**.
- La lecture, la division et l'interprétation des données d'entrée sont gérées par une classe appelée **InputFormat**.

Classe InputFormat

La classe **InputFormat** influence :

- La **récupération** des données sources : HDFS, bases de données, etc.
- La **division** des données d'entrée par la classe **InputSplit**.
- La **lecture** des données d'entrée par la classe **RecordReader**.
- Les **types** de couples clef/valeur passées au **Mapper** (à partir des données d'entrée lues).

Classe InputFormat

- Hadoop fournit plusieurs classes **InputFormat**.
- Pour utiliser une classe **InputFormat** différente au lieu de celle par défaut :

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

- La classe utilisé par **défaut** est :

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

- Le comportement peut souvent être influencé (par l'objet de configuration ou des méthodes statiques).

Classe InputFormat - Exemples

KeyValueTextInputFormat :

- Découpe le ou les fichiers d'entrée par ligne.
- S'attend à trouver des couples (clef; valeur) au sein de chacune des lignes.
- Le séparateur par défaut pour la clé et la valeur est le caractère de tabulation.
- Exemple :

```
1    2,5|GRIS|0  
2    3|BLANC|-1
```

- Le caractère utilisé pour la séparation peut être modifié ainsi :

```
conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", ";")
```

Classe InputFormat - Exemples

FixedLengthInputFormat :

- Découpe le ou les fichiers d'entrée selon une taille fixe indiquée.
- La **clef** est le numéro du fragment, la **valeur** est le fragment lui-même.
- Particulièrement adapté pour des données binaires où une série de « blocs » de taille égale se suivent.
- Exemple : lire le code RNA par codon (trois caractères).

```
GCUACGGAGCUUCGGAGCUAGGCUACGGAGCUUCGGAGCUAG
```

- La taille du bloc peut être réglée par les paramètres suivants :

```
conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, 3);
```

```
FixedLengthInputFormat.setRecordLength(conf, 3);
```


Classe InputFormat - Exemples

NLineInputFormat :

- Découpe le ou les fichiers d'entrée par groupes de N lignes.
- La clef est le numéro du bloc de N lignes.
- La valeur est le contenu des lignes.
- Exemple : lire par groupes de deux lignes.

Il s'agit d'une
phrase de deux lignes
Et voici
une autre

- Le nombre N des lignes peut être réglée par les paramètres suivants :

```
conf.setInt("mapreduce.input.lineinputformat.linespermap", 2);
```

```
NLineInputFormat.setNumLinesPerSplit(job, 2);
```

Classe InputFormat - Exemples

Sequence Files :

- Format de Apache destiné à stocker des couples clé-valeur.
- Supporte le **format binaire**.
- Supporte la **compression**.
- Plus efficace ; évite les problèmes de surcharge avec de nombreux petits fichiers.
- Souvent utilisé avec Hadoop et les frameworks big data en général.

Classe InputFormat - Exemples

SequenceFileInputFormat :

- Lit les fichiers de type "Sequence File".
- Les clés et les valeurs sont extraites du fichier de séquence; leurs types de données varient en fonction du fichier.
- Souvent utilisé en production, bien que le format textuel soit également courant.
- Les fichiers de séquence sont couramment créés avec les programmes Hadoop; souvent par le biais d'une opération map uniquement : pas de shuffle, pas de reduce, les tuples produit par map sont écrits directement en sortie. Peut être accompli avec :

```
job.setNumReduceTasks(0);
```

Classe InputFormat

- De nombreuses autres classes **InputFormat** sont disponibles.
- Ils ne sont pas tous basés sur des fichiers HDFS; par exemple : **MongoInputFormat** avec le **connecteur MongoDB**.
- Pour des cas d'utilisation particuliers, il est possible de créer son propre InputFormat.

Créer son propre InputFormat

Pour créer **son propre InputFormat** :

- La nouvelle classe doit hériter d'un InputFormat existant : **FileInputFormat**, **DBInputFormat**, **CompositeInputFormat** ...
- Créer une classe héritant de **RecordReader**.
- (Ré)Implémenter les méthodes pertinentes dans les deux.

Créer son propre InputFormat - Exemple

Exemple :

- Un InputFormat pour lire les données de notre exemple "amis communs" comme couples clef/valeur appropriés dès le départ.
- Pour rappel, les données d'entrée sont :

```
A => B, C, D  
B => A, C, D, E  
C => A, B, D, E  
D => A, B, C, E  
E => B, C, D
```

Créer son propre InputFormat - Exemple

Objets Hadoop principaux pour travailler avec des données d'entrée provenant de fichiers :

- **InputSplit** est un bloc qui contient des données; il est renvoyé par la méthode getSplits() de l'InputFormat. Pour le format FileInputFormat, par défaut: InputSplit \sim un block sur HDFS.
- **RecordReader** lit InputSplit ; génère des tuples à partir de lui.
- **InputFormat** encapsule et crée les deux.

Créer son propre InputFormat - Exemple

→ Notre propre classe **InputFormat** pour l'exemple "amis communs" :

```
public class FriendsInputFormat extends FileInputFormat<Text, Text> {  
    public RecordReader<Text, Text> createRecordReader(InputSplit split, TaskAttemptContext  
context)  
    throws IOException, InterruptedException {  
        return new FriendsRecordReader();  
    }  
}
```


Créer son propre InputFormat - Exemple

→ Notre propre classe **RecordReader** pour l'exemple "amis communs" : (1/3)

```
public class FriendsRecordReader extends RecordReader<Text, Text> {  
    private LineRecordReader lineRecordReader = null;  
    private Text key = null;  
    private Text value = null;  
  
    public void initialize(InputSplit split, TaskAttemptContext context)  
        throws IOException, InterruptedException {  
        close();  
        lineRecordReader = new LineRecordReader();  
        lineRecordReader.initialize(split, context);  
    }  
}
```

Créer son propre InputFormat - Exemple

→ Notre propre classe **RecordReader** pour l'exemple "amis communs" : (2/3)

```
public boolean nextKeyValue() throws IOException, InterruptedException {
    if(!lineRecordReader.nextKeyValue()) {
        key = null;
        value = null;
        return false;
    }
    Text line = lineRecordReader.getCurrentValue();
    String str = line.toString();
    String[] arr = str.split("=>");
    key = new Text(arr[0].trim());
    value = new Text(arr[1].trim());
    return true;
}

public Text getCurrentKey() throws IOException, InterruptedException {
    return key;
}
```

Créer son propre InputFormat - Exemple

→ Notre propre classe **RecordReader** pour l'exemple "amis communs" : (3/3)

```
public Text getCurrentValue() throws IOException, InterruptedException {  
    return value;  
}  
public float getProgress() throws IOException, InterruptedException {  
    return lineRecordReader.getProgress();  
}  
public void close() throws IOException {  
    if(lineRecordReader != null) {  
        lineRecordReader.close();  
        lineRecordReader = null;  
    }  
    value = null;  
}
```

Créer son propre InputFormat - Exemple

Remarques :

- Nous avons accès à la configuration par le biais du contexte ; nous pouvons permettre à l'utilisateur final d'ajuster le comportement de notre **InputFormat** en utilisant des variables de configuration.
- Nous utilisons **LineRecordReader** : le **RecordReader** du **InputFormat** par défaut - **TextInputFormat**.
- C'est l'une des approches les plus courantes ; elle nous permet de lire facilement par ligne. Alternativement : il faut mettre en œuvre la lecture de l'**InputSplit** manuellement, par exemple avec l'**API HDFS**.

Créer son propre InputFormat - Exemple

→ Un exemple d'initialisation pour la gestion manuelle d'**InputSplit** :

```
public void initialize(InputSplit isplit, TaskAttemptContext ctx) throws IOException {  
    FileSplit split = (FileSplit) isplit;  
    Configuration conf = context.getConfiguration();  
    start = split.getStart();  
    end = start + split.getLength();  
    Path file = split.getPath();  
    FileSystem fs = file.getFileSystem(job);  
    fileIn = fs.open(file);  
    fileIn.seek(start);  
}
```

Créer son propre InputFormat - Exemple

→ Dans la même classe, pour lire une ligne et une clef/valeur, nous ferions par exemple :

```
public boolean nextKeyValue() throws IOException {  
    // ...  
    Text line = new Text();  
    size = fileIn.readLine(line, MAX_LENGTH);  
    String[] arr = line.toString().split("=>");  
    key = new Text(arr[0].trim());  
    value = new Text(arr[1].trim());  
    // ...  
}
```

Créer son propre InputFormat - Exemple

Et concernant les **(input) splits** :

- Le comportement peut être modifié en surchargeant **getSplits()** dans **InputFormat**.
- Renvoie une liste d'objets **InputSplit**.
- Rarement nécessaire pour les formats d'entrée de fichiers (le comportement par défaut convient généralement).

Classe OutputFormat

- Équivalent de la classe InputFormat, mais pour les données de sortie.
- Comportement par défaut : un tuple de sortie par ligne, clef puis tabulation puis valeur.
- Peut être modifié de la même manière que le format d'entrée :

```
job.setOutputFormatClass(class);
```

- Par exemple (classe de format de sortie par défaut) :

```
job.setOutputFormatClass(TextOutputFormat.class);
```


Classe OutputFormat

Il existe une multitude de classes standard **OutputFormat** fournis par Hadoop :

- **SequenceFileOutputFormat** : écrit des fichiers de séquences.
- **MultipleOutputFormat** : écrit les couples clef/valeur de sortie dans différents emplacements sur HDFS en fonction des clés ou des valeurs des tuples.
- Il y en a d'autres, leur comportement peut également être modifié par le biais des paramètres de configuration.

Classe **OutputFormat**

- Par exemple, pour ajuster le séparateur dans la classe **TextOutputFormat** par défaut (pour utiliser le point-virgule au lieu de la tabulation) :

```
conf.set("mapreduce.output.textoutputformat.separator", ";");
```

- Comme pour l'**InputFormat**, pour des besoins plus complexes, nous pouvons créer notre propre **OutputFormat**.
- Approche très similaire : il suffit de créer les classes **OutputFormat** et **RecordWriter**.

Créer son propre OutputFormat

L'**OutputFormat** doit principalement :

- Ouvrir un fichier de sortie sur HDFS pour les résultats d'une opération Reduce.
- Instancier et retourner un objet RecordWriter.

Le **RecordWriter** écrit des couples clef/valeur dans le fichier de sortie.

Créer son propre OutputFormat - Exemple

Exemple :

- Nous voulons maintenant créer notre **propre OutputFormat** pour notre exemple "amis communs".
- Nous voulons que les données finales soient écrites comme suit :

```
A-B => C, D  
A-C => B, D  
A-D => B, C  
B-C => A, D, E  
B-D => A, C, E  
...
```

Créer son propre OutputFormat - Exemple

→ Notre classe **OutputFormat** :

```
public class FriendsOutputFormat extends FileOutputFormat<Text, Text> {  
    public RecordWriter<Text, Text> getRecordWriter(TaskAttemptContext context)  
    throws IOException, InterruptedException {  
        Path path = getDefaultWorkFile(context, "");  
        FileSystem fs = path.getFileSystem(context.getConfiguration());  
        FSDataOutputStream fileOut = fs.create(path, context);  
        return(new FriendsRecordWriter(fileOut));  
    }  
}
```

Créer son propre OutputFormat - Exemple

→ Notre classe **RecordWriter** :

```
public class FriendsRecordWriter extends RecordWriter<Text, Text> {  
    private DataOutputStream out;  
  
    public FriendsRecordWriter(DataOutputStream stream) {  
        out = stream;  
    }  
    public void write(Text k, Text val) throws IOException, InterruptedException {  
        out.writeBytes(k.toString() + " => " + val.toString() + "\n");  
    }  
    public void close(TaskAttemptContext ctx) throws IOException,  
        InterruptedException {  
        out.close();  
    }  
}
```

Créer son propre OutputFormat - Exemple

→ Nous pouvons aussi, bien sûr, **modifier le nom du fichier de sortie** :

```
public class FriendsOutputFormat extends FileOutputFormat<Text, Text> {  
    public RecordWriter<Text, Text> getRecordWriter(TaskAttemptContext context)  
        throws IOException, InterruptedException {  
        Path path = FileOutputFormat.getOutputPath(context);  
        Path fullPath = new Path(  
            path, FileOutputFormat.getUniqueFile(context, "RESULTS", ".txt")  
        );  
        FileSystem fs = path.getFileSystem(context.getConfiguration());  
        FSDataOutputStream fileOut = fs.create(fullPath, context);  
        return(new FriendsRecordWriter(fileOut));  
    }  
}
```

Créer son propre OutputFormat - Exemple

→ Avant notre changement de nom, le répertoire de sortie sur HDFS ressemblait à ceci :

```
/out/_SUCCESS  
/out/part-r-00000  
/out/part-r-00001  
/out/part-r-00002  
...
```

→ Après:

```
/out/_SUCCESS  
/out/RESULTS-r-00000.txt  
/out/RESULTS-r-00001.txt  
/out/RESULTS-r-00002.txt  
...
```


Les types Writable

- Jusqu'ici, on a utilisé des types « simples » pour les clefs et valeurs dans les exemples : Text ou encore IntWritable.
- Hadoop permet également de définir ses **propres types Writable**, pour la **clef** et la **valeur**.
- Si nous souhaitons utiliser nos propres types comme entrée initiale ou comme sortie finale du programme, il est nécessaire d'implémenter un **InputFormat** ou, respectivement, un **OutputFormat** dédié.

Les types Writable

Deux options :

- Si notre type serait utilisé **uniquement comme valeur** des tuples => implémenter l'interface **Writable**.
- Si notre type serait utilisé comme **clef et valeur** des tuples => implémenter l'interface **WritableComparable**.

Dans les deux cas, la sérialisation et la désérialisation doivent être gérées par la classe spécifique.

Les types Writable - Exemple

Exemple :

- On souhaite créer un type **Writable** pour stocker nos valeurs dans l'exemple « amis en commun ».
- La classe s'appellera « FriendsListWritable » et stockera la liste des amis liés à un utilisateur du réseau.

Les types Writable - Exemple

→ Exemple d'un type Writable spécifique (1/2) :

```
public class FriendsListWritable implements Writable {  
    public ArrayList<String> friends;  
  
    public void write(DataOutput out) throws IOException {  
        Text line = new Text(get_serialized());  
        line.write(out);  
    }  
    public FriendsListWritable(String datatxt) {  
        unserialize(datatxt);  
    }  
    public void unserialize(String datatxt) {  
        String[] data = datatxt.split(",");  
        friends = new ArrayList<String>(Arrays.asList(data));  
    }  
}
```

Les types Writable - Exemple

→ Exemple d'un type Writable spécifique (2/2) :

```
public FriendsListWritable() { }

public String get_serialized() {
    String line = String.join(",", friends);
    return(line);
}

public void readFields(DataInput in) throws IOException {
    Text line = new Text();
    line.readFields(in);
    unserialize(line.toString());
}
}
```

Les types Writable - Exemple

→ Pour utiliser notre type comme valeur de sortie, spécifions le **dans le Driver** :

```
job.setInputFormatClass(FriendsInputFormat.class);  
job.setOutputFormatClass(FriendsOutputFormat.class);  
job.setOutputValueClass(FriendsListWritable.class);
```

→ Et **dans le Mapper / Reducer** :

```
public class FriendsMap extends Mapper<Text, FriendsListWritable, Text, FriendsListWritable>
```

```
public class FriendsReduce extends Reducer<Text, FriendsListWritable, Text,  
FriendsListWritable>
```

Les types Writable - Exemple

→ Notre **InputFormat** doit également être mis à jour :

```
public class FriendsInputFormat extends FileInputFormat<Text, FriendsListWritable> {
```

→ Le **RecordReader** aussi :

```
public boolean nextKeyValue() throws IOException, InterruptedException {  
    // ...  
    String[] arr = str.split("=>");  
    key = new Text(arr[0].trim());  
    value = new FriendsListWritable(arr[1].trim());  
    // ...  
}
```

Les types Writable - Exemple

- Pour utiliser le nouveau type Writable comme **clef** également, implémentez **WritableComparable**.
- Deux méthodes supplémentaires à implémenter :
 - ◆ **compareTo()** :
similaire à la fonction de comparaison habituelle de Java ; compare deux instances.
 - ◆ **hashCode()** :
doit renvoyer une valeur dont la similitude est garantie pour deux objets identiques - même à travers des JVM et différentes exécutions.

Les types Writable - Example

- Le code de hachage est utilisé pendant l'étape **shuffle**, pour produire les partitions qui sont passées à l'étape **reduce**.
- Deux hashcodes identiques permettent au **shuffle** de regrouper des objets identiques (clefs identiques).
- Les objets Java disposent d'une fonction `hashCode()` ; cependant, d'après la doc Java : Cet hashcode n'a pas besoin de rester cohérent d'une exécution d'une application à une autre exécution de la même application.

Les types Writable - Example

- Comme nous exécutons notre programme sur plusieurs machines, la fonction hashCode() sera appelée sur des objets identiques mais sur des machines différentes.
- Par conséquent, bien qu'elles soient égales, les clefs ne seront peut-être pas regroupées.
- **Solution** : générer nous-mêmes un hashCode, qui soit **basé sur les données de l'objet**. De plus, le hashCode doit être **distribué uniformément**.
- Les types **Writable** Hadoop standard (Text, etc.) offrent déjà une telle fonction de hashCode.

Les types Writable - Example

→ Pour revenir à notre exemple, nous pourrions faire :

```
public class FriendsListWritable implements  
WritableComparable<FriendsListWritable> {  
    // ...
```

→ Et :

```
    public int hashCode() {  
        return(new Text(get_serialized()).hashCode());  
    }  
    public int compareTo(FriendsListWritable o) {  
        int mysize = friends.size();  
        int theirsiz = o.friends.size();  
        return (mysize < theirsiz ? -1 : (mysize == theirsiz ? 0 : 1));  
    }  
    // ...
```