MBDS

Université Nice SOPHIA ANTIPOLIS

Hadoop / Big Data

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MBDS

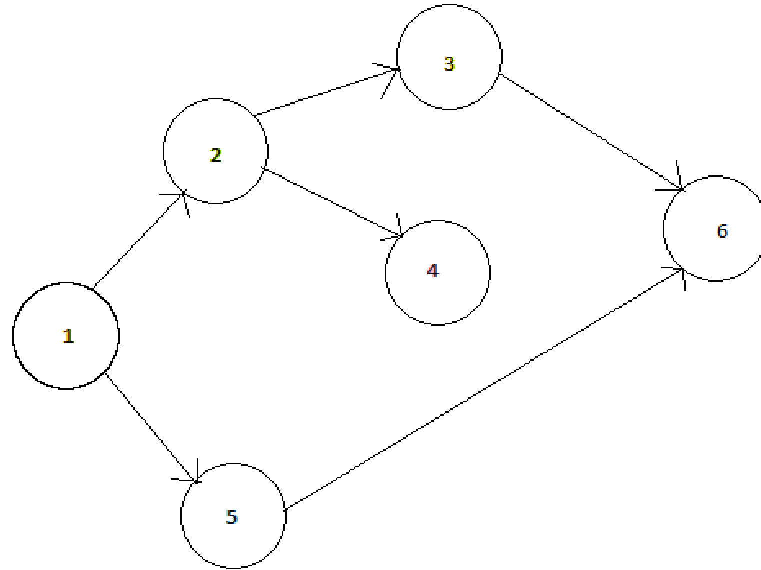
2019 - 2020

6 map/reduce et Hadoop: exemples plus avancés

Exemple: parcours de graphe

- **On cherche à déterminer la profondeur maximale de tous les nœuds d'un graphe à partir d'un nœud de départ (via *breadth-first search*, parcours en largeur).**

- **Le graphe:**



Exemple: parcours de graphe

- **Données d'entrée:**

```
(1;"2,5|GRIS|0")
(2;"3,4|BLANC|-1")
(3;"6|BLANC|-1")
(4;"|BLANC|-1")
(5;"6|BLANC|-1")
(6;"|BLANC|-1")
```

… avec valeur sur le modèle « NOEUDS_FILS|COULEUR|PROFONDEUR ».

- **Et couleur ayant pour valeur « BLANC » pour un nœud non parcouru, « GRIS » pour un nœud en cours de parcours, et « NOIR » pour un nœud déjà parcouru. Dans nos données d'entrée, l'unique nœud GRIS indique le nœud de départ.**

Exemple: parcours de graphe

- *map*: si le nœud du couple (clef;valeur) d'entrée est GRIS, alors:
  - Pour chacun de ses fils, retourner (ID_FILS;"|GRIS|PROFONDEUR+1), où PROFONDEUR est la profondeur du nœud courant.
  - Retourner également le couple (clef;valeur) courant, avec couleur=NOIR.
- Sinon: retourner le couple (clef;valeur) d'entrée.

- Pseudo code:

```
SI NODE.COULEUR=="GRIS":
  POUR CHAQUE FILS DANS NODE.CHILDREN:
    FILS.COULEUR="GRIS"
    FILS.PROFONDEUR=NODE.PROFONDEUR+1
    RENVOYER (FILS.ID;FILS)
  NODE.COULEUR="NOIR"
RENVOYER (NODE.ID;NODE)
```

Exemple: parcours de graphe

- ***reduce*: parcourir chacune des valeurs associées à la clef unique (après *shuffle*). Renvoyer un couple (clef;NOEUD) avec un nœud dont:**

  - **La profondeur est la plus haute rencontrée parmi les valeurs associées à cette clef unique (l'identifiant du nœud).**
  - **La couleur est la plus « forte » parmi ces mêmes valeurs.**
  - **La liste des nœuds enfants est la plus longue parmi ces valeurs.**

  … et pour clef la clef unique en question.

Exemple: parcours de graphe

- Pseudo code:

```
H_CHILDREN=""; H_PROF=-1; H_COULEUR="BLANC";

POUR CHAQUE VALEUR:
  SI VALEUR.CHILDREN.LENGTH()>H_CHILDREN.LENGTH():
    H_CHILDREN=VALEUR.CHILDREN
  SI VALEUR.PROFONDEUR>H_PROF:
    H_PROF=VALEUR.PROFONDEUR
  SI VALEUR.COULEUR>H_COULEUR:
    H_COULEUR=VALEUR.COULEUR

NODE=NOUVEAU NOEUD
NODE.COULEUR=H_COULEUR
NODE.CHILDREN=H_CHILDREN
NODE.PROFONDEUR=H_PROF
RENVOYER(CLEF;NODE)
```

Exemple: parcours de graphe

- **On exécute le programme map/reduce plusieurs fois, jusqu'à ce que tous les nœuds de la liste aient pour couleur la valeur « NOIR » <=> jusqu'à ce que tous les nœuds aient été parcourus.**

- **Ce type de logique s'implémente très facilement au sein d'un *framework* map/reduce (Hadoop).**

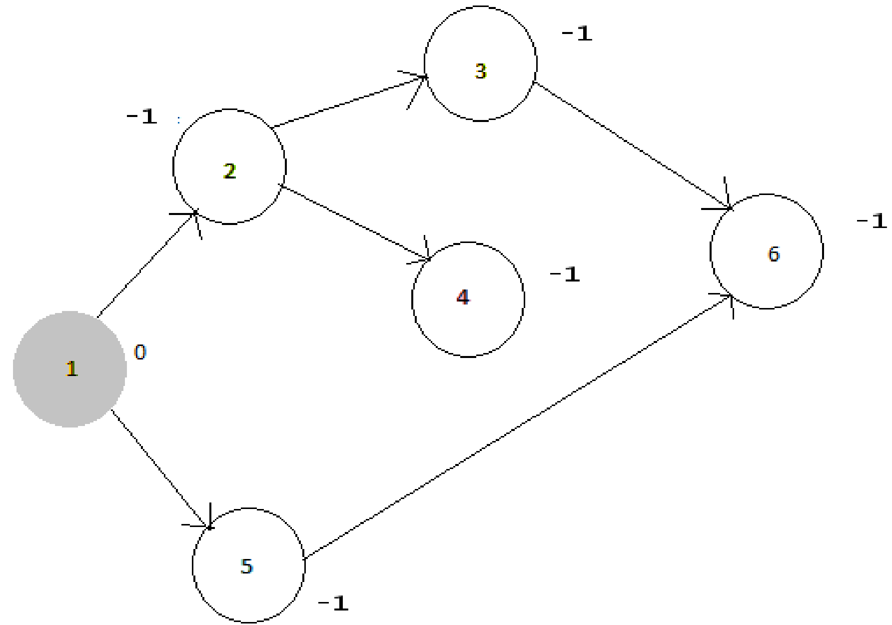
Exemple: parcours de graphe

- **Premier lancement:**

Données d'entrée:

```
(1;"2,5|GRIS|0")
(2;"3,4|BLANC|-1")
(3;"6|BLANC|-1")
(4;"|BLANC|-1")
(5;"6|BLANC|-1")
(6;"|BLANC|-1")
```

Graphe:



Exemple: parcours de graphe

- **Pour ce premier lancement:**

Sortie de *map*:

```
(1;"2,5|NOIR|0")
(2;"|GRIS|1")
(5;"|GRIS|1")
(2;"3,4|BLANC|-1")
(3;"6|BLANC|-1")
(4;"|BLANC|-1")
(5;"6|BLANC|-1")
(6;"|BLANC|-1")
```

Aprés *shuffle*:

```
1:  ("2,5|NOIR|0")
2:  ("|GRIS|1"),("3,4|BLANC|-1")
3:  ("6|BLANC|-1")
4:  ("|BLANC|-1")
5:  ("|GRIS|1"),("6|BLANC|-1")
6:  ("|BLANC|-1")
```

Sortie de *reduce*:

```
(1;"2,5|NOIR|0")
(2;"3,4|GRIS|1")
(3;"6|BLANC|-1")
(4;"|BLANC|-1")
(5;"6|GRIS|1")
(6;"|BLANC|-1")
```

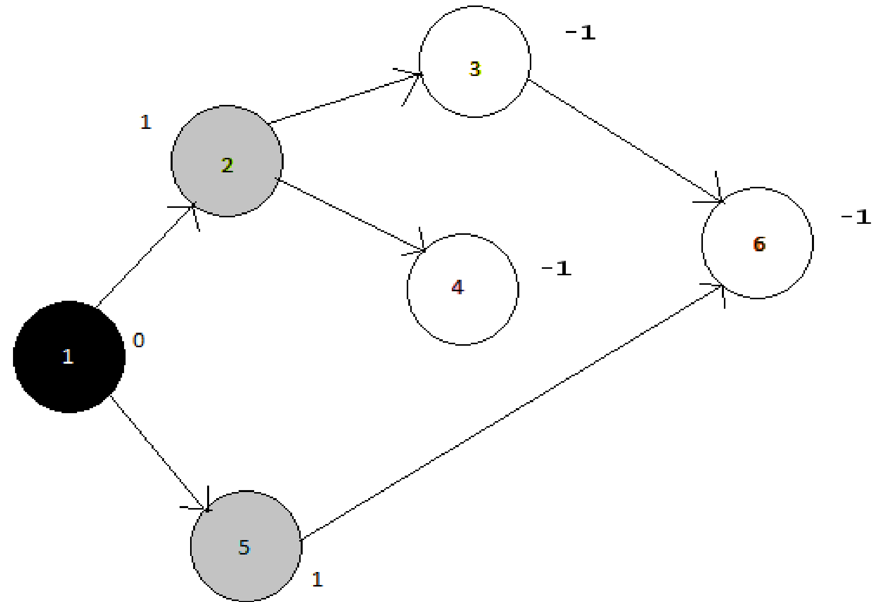
Exemple: parcours de graphe

- **Lancement 2:**

Données:

```
(1;"2,5|NOIR|0")
(2;"3,4|GRIS|1")
(3;"6|BLANC|-1")
(4;"|BLANC|-1")
(5;"6|GRIS|1")
(6;"|BLANC|-1")
```

Graphe:



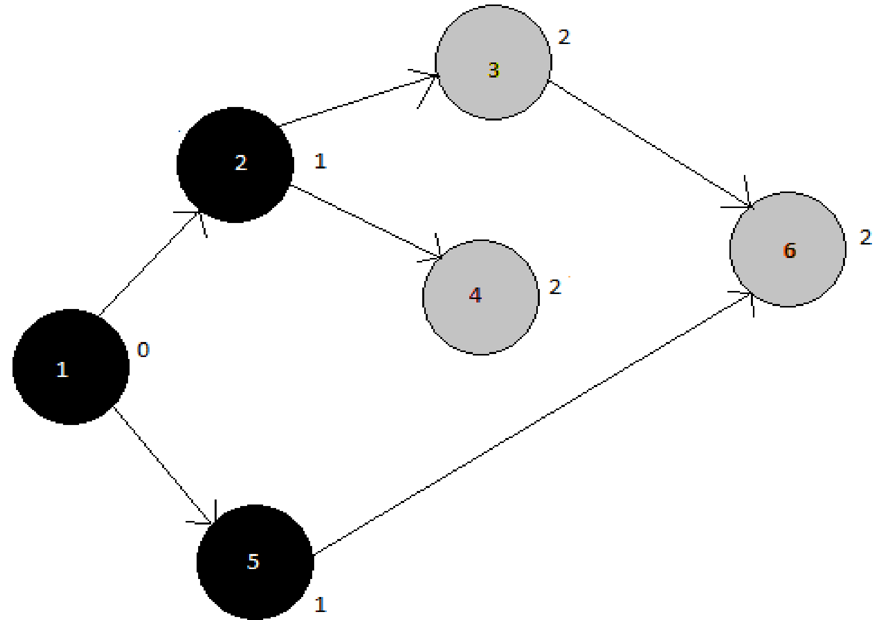
Exemple: parcours de graphe

- **Lancement 3:**

Données:

```
(1;"2,5|NOIR|0")
(2;"3,4|NOIR|1")
(3;"6|GRIS|2")
(4;"|GRIS|2")
(5;"6|NOIR|1")
(6;"|GRIS|2")
```

Graphe:



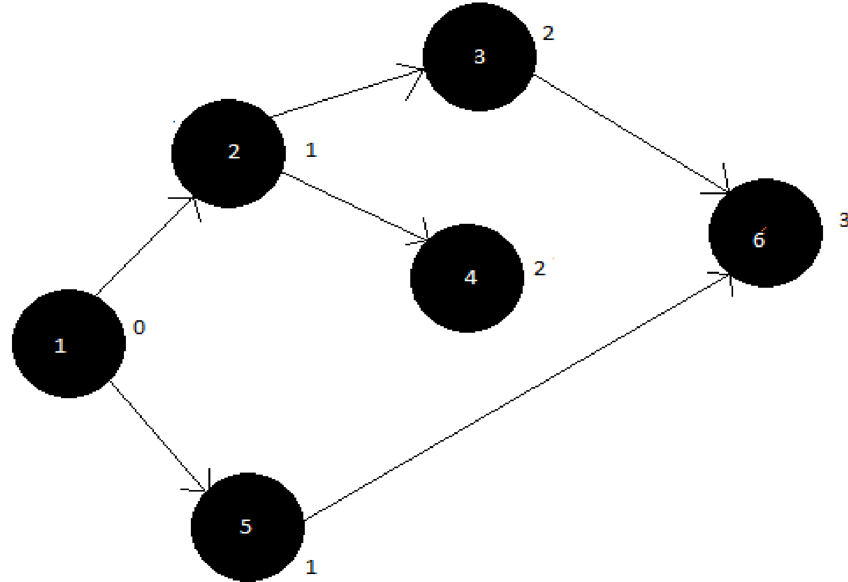
Exemple: parcours de graphe

- **Après le lancement 3:**

Données de sortie:

```
(1;"2,5|NOIR|0")
(2;"3,4|NOIR|1")
(3;"6|NOIR|2")
(4;"|NOIR|2")
(5;"6|NOIR|1")
(6;"|NOIR|3")
```

Graphe:



Exemple: parcours de graphe

- **Tous les nœuds ont tous désormais pour couleur la valeur « NOIR » ; le programme en charge de lancer la tâche map/reduce à répétition sur les données s’arrête.**

- **On a ainsi effectué un parcours en largeur sur le graphe, parallélisable (*parallel breadth-first search*). A chaque étape de profondeur dans le parcours correspond l'exécution d'une tâche map/reduce – avec une fonction *map* executée pour chaque nœud à chaque étape.**

Conclusion

- **Applications courantes de map/reduce et Hadoop:**

  - **Analyse de *logs* et données en général, validation de données, recoupements, filtrage, etc. => traitements sur des volumes de données massifs.**
  - **Exécution de tâches intensives en CPU de manière distribuée (simulations scientifiques, encodage vidéo, etc.).**

… et bien souvent les deux: tâches intensives en CPU sur des volumes de données massifs (entraînement de modèles en *machine learning*, etc.)

- **Plus complexe, voire contre-productif à appliquer sur tout problème où la fragmentation des données d'entrée pose problème (en fait sur tout problème où la stratégie du *diviser pour régner* n'est pas viable).**

7 map/reduce et Hadoop: développement plus avancé

Les InputFormats Hadoop

- **Lors de l'exemple du compteur d'occurence de mots vu précédemment, Hadoop ouvre le fichier d'entrée (le poème) sur HDFS et le découpe automatiquement; en générant un fragment par ligne.**

- **Cette logique provient du fait que par défaut, c'est le comportement appliqué par Hadoop: l'entrée est au format texte, et chaque ligne dénote un fragment distinct; et Hadoop émet en entrée de map des couples (clef;valeur) pour lesquels la clef est le numéro de la ligne dans le fichier, et la valeur le contenu de la ligne elle-même.**

- **Ce comportement de lecture des données d'entrée peut être influencé par le biais de classes implémentant un comportement différent: les `InputFormats`.**

Les InputFormats Hadoop

- **Un InputFormat a deux principales responsabilités:**
  - **S'assurer du découpage cohérent des données situées dans un fichier d'entrée.**
  - **Fournir une logique d'interprétation des données d'entrée pour générer des couples (clef;valeur) cohérents.**
- **La classe InputFormat utilisée par défaut est TextInputFormat.**
- **Il est possible d'en sélectionner une autre lors de la configuration de la tâche, et même de créer son propre InputFormat spécifique.**

Selection d'un InputFormat

- **Pour sélectionner une classe InputFormat différente, on utilisera la méthode:**

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

  (de l'objet job désignant la tâche à exécuter dans la classe driver)

- **Par exemple:**

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

  … pour spécifier la classe InputFormat KeyValueTextInputFormat.

Quelques InputFormats

- **Quelques exemples d'InputFormats Hadoop alternatifs standards:**

  - **`KeyValueTextInputFormat`**
    Découpe le ou les fichiers d'entrée par ligne; mais contrairement au TextInputFormat utilisé par défaut, cette classe s'attend à trouver des couples (clef;valeur) au sein de chacune des lignes, sur un format constitué d'une clef, puis d'une tabulation, puis du reste de la ligne comme valeur. Ces couples (clef;valeur) seront émis à la fonction map en conséquence.

  - **`FixedLengthInputFormat`**
    Découpe le ou les fichiers d'entrée selon une taille fixe indiquée à la classe. Les couples (clef;valeur) émis ont le numéro du fragment dans le fichier pour clef, et le fragment lui-même comme valeur. Particulièrement adapté à des données binaires où une série de « blocs » de taille égale se suivent.

Quelques InputFormats

- **`NLineInputFormat`**
  Découpe le ou les fichiers d'entrée par groupes de N lignes; la clef transmise à la fonction map représente le numéro du bloc de N lignes, et la valeur le contenu des lignes.

- **`SequenceFileInputFormat`**
  Un InputFormat adapté aux Sequence Files; il s'agit d'un format de données binaire communément utilisé dans Hadoop, des outils associés à Hadoop, et d'une manière générale dans des logiciels Big Data. Le format est adapté à une lecture rapide, et est souvent préféré aux fichiers texte dans des applications de production.

Configurer un InputFormat

- **La plupart des InputFormat Hadoop supportent des paramètres de configuration pour influer sur leur fonctionnement. Ces paramètres sont généralement ajustables par le biais d'un appel à des méthodes:**

  ```
  conf.set() / conf.setInt() / conf.setLong() / ...
  ```

  (de l'objet Configuration associé à la tâche au sein de la classe driver)

- **Par exemple:**

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

  Pour indiquer à l'InputFormat `KeyValueInputFormat` que le séparateur entre la clef et la valeur dans les données d'entrée n'est pas une tabulation mais un caractère « : ».

Configurer un InputFormat

- **Un autre exemple:**

  ```
  conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, 12);
  ```

  Pour indiquer à l'InputFormat `FixedLengthInputFormat` que la taille de chacun des blocs fixes de données à lire est de 12 octets.

- **Certains InputFormat sont configurables / doivent être configurés par le biais d'une méthode statique sur la classe InputFormat plutôt que d'un appel à une des méthodes `set` de l'objet Configuration.**

- **Remarque: la configuration doit être effectuée avant l'instanciation de l'objet Job (qui récupère la configuration passée en paramètre).**

Créer son propre InputFormat

- **Tous les InputFormats vus précédemment héritent d'une classe « FileInputFormat », qui défini un InputFormat lisant des données depuis un ou des fichiers sur HDFS.**

- **Il en existe d'autres: `DBInputFormat` pour lire des données depuis une base de données, `CompositeInputFormat` pour lire des données depuis plusieurs sources à la fois, etc. certains exemples de ces InputFormat seront vus plus loin dans le cours; c'est entre autres leur existence qui permet d'intégrer Hadoop à d'autres logiciels (par exemple une base de données NoSQL en entrée d'un programme map/reduce, sans passer par HDFS).**

- **Il est également possible de créer son propre InputFormat; par exemple pour lire des données dans un format propriétaire de manière cohérente depuis HDFS, ou encore pour obtenir des couples (clef;valeur) depuis une API tierce propres à un développement spécifique, etc.**

Créer son propre InputFormat

- **Pour créer son propre FileInputFormat, par exemple, on suivra la procédure suivante:**

  - **Créer une classe qui hérite de `FileInputFormat`, en paramétrisant cette classe avec deux types: les types de clef et valeur susceptibles d'être lus *via* cet InputFormat.**

  - **Créer une classe héritant de `RecordReader`, paramétrisée de la même manière, et qui aura pour tâche de lire les différents couples (clef;valeur) au sein de chacun des fichiers d'entrée.**

  - **Implémenter plusieurs méthodes au sein de ces deux classes afin de créer la logique de lecture des fichiers d'entrée.**

Créer son propre InputFormat - Exemple

- **Imaginons qu'on souhaite lire des fichiers source correspondant à l'exemple des « amis en commun » du réseau social vu précédemment.**

- **Ces fichiers respectent le format suivant:**

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

On devra donc implémenter un FileInputFormat capable d'interpréter chacune de ces lignes pour séparer la clef (l'utilisateur concerné) de la valeur (la liste d'amis), en identifiant le séparateur (les caractères « => »).

Créer son propre InputFormat - Exemple

- **Le code de la classe InputFormat:**

```
package org.mbds.hadoop.friends;

import …

// Notre InputFormat spécifique.
public class FriendsInputFormat extends FileInputFormat<Text, Text> {

    // La méthode de création d'un RecordReader, à implémenter.
    public RecordReader<Text, Text> createRecordReader(InputSplit split,
   TaskAttemptContext context) throws IOException, InterruptedException {
       // Renvoie simplement une instance de notre RecordReader
       // spécifique.
       return new FriendsRecordReader();
    }
}
```

Créer son propre InputFormat - Exemple

- **Le code de la classe RecordReader:**

```
package org.mbds.hadoop.friends;

import …

// Notre RecordReader spécifique.
public class FriendsRecordReader extends RecordReader<Text, Text> {
    private LineRecordReader lineRecordReader=null;
    private Text key=null;
    private Text value=null;

     // Fermeture du reader.
    public void close() throws IOException {
        if(lineRecordReader!=null)
        {
            lineRecordReader.close();
            lineRecordReader=null;
```

Créer son propre InputFormat - Exemple


```
        }
        key=null;
        value=null;
    }

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

public Text getCurrentValue() throws IOException, InterruptedException {
        return value;
    }

    // Pour renvoyer la progression actuelle de la lecture.
    public float getProgress() throws IOException, InterruptedException {
        return lineRecordReader.getProgress();
    }
```

Créer son propre InputFormat - Exemple


```
// Initialisation.
 public void initialize(InputSplit split, TaskAttemptContext context)
  throws IOException, InterruptedException {
     close();
     lineRecordReader=new LineRecordReader();
     lineRecordReader.initialize(split, context);
 }

// Fonction principale qui a la charge de la lecture du couple
// (clef;valeur) suivant au sein du flux.
 public boolean nextKeyValue() throws IOException, InterruptedException
     if(!lineRecordReader.nextKeyValue()) {
         key = null;
         value = null;
         return false;
     }
```

Créer son propre InputFormat - Exemple


```
        // Logique de lecture.
        // On récupère simplement la ligne lue depuis le fichier par
        // le LineRecordReader; et on sépare clef et valeur.
        // Les clef et valeur ne sont pas retournées ici, mais simplement
        // lues et stockées.
        Text line=lineRecordReader.getCurrentValue();
        String str=line.toString();
        String[] arr=str.split("=>");
        key=new Text(arr[0]);
        value=new Text(arr[1]);
        return true;
    }
}
```

Créer son propre InputFormat

- Il ne s'agit là que d'un rapide tour d'horizon de l'implémentation d'un InputFormat spécifique; il va de soit que des possibilités plus complexes sont disponibles.

- Pour plus d'informations, se référer à la documentation et aux exemples Hadoop:

  https://hadoop.apache.org/docs/

Les OutputFormats Hadoop

- **De la même manière que pour les InputFormats, on peut également influencer la façon dont Hadoop va écrire les résultats finaux, c'est à dire les couples (clef;valeur) issus de l'opération Reduce, sur HDFS.**

- **Pour ce faire, on procède de la même manière, mais à l'aide d'un autre type de classes: les OutputFormats.**

- **La méthode permettant d'indiquer à Hadoop qu'on souhaite utiliser un OutputFormat spécifique est la suivante:**

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

(de l'objet Job désignant la tâche à effectuer dans la classe driver)

Les OutputFormats Hadoop

- **L'OutputFormat par défaut de Hadoop est la classe `TextOutputFormat`. Elle produit un format similaire au format d'entrée de `KeyValueInputFormat`, c'est à dire:**

  - **Un couple (clef;valeur) par ligne.**
  - **Une tabulation entre la clef et la valeur.**

- **C'est pour cette raison que lors de l'exécution des programmes en TP, les fichiers de résultat contenaient nos couples (clef;valeur) finals sous ce format.**

- **Comme pour les InputFormats, Hadoop propose plusieurs OutputFormat standards.**

Quelques OutputFormats

- **Quelques exemples d'OutputFormats Hadoop alternatifs standards:**
  - **`SequenceFileOutputFormat`**
    Écrit les couples (clef;valeur) sur HDFS sous la forme d'un *Sequence File* – le même format binaire que celui qui est utilisé avec la classe `SequenceFileInputFormat`.
  - **`MultipleOutputFormat`**
    Écrit les couples (clef;valeur) vers plusieurs destination; cette classe est le pendant de l'InputFormat MultipleInputFormat et permet par exemple d'écrire les couples (clef;valeur) à plusieurs endroits différents sur HDFS.

Les OutputFormats Hadoop

- Comme pour les InputFormats, ces classes sont issues d'une classe mère FileOutputFormat; et là aussi, il en existe des variantes, par exemple DBOutputFormat pour écrire les couples (clef;valeur) vers des bases de données.

- Là aussi, le comportement des OutputFormats est configurable. Par exemple, en utilisant:

```
conf.set("mapreduce.output.textoutputformat.separator", ";");
// ...
job.setOutputFormatClass(TextOutputFormat.class);
```

  … on indiquera qu'on souhaite utiliser le séparateur « : » plutôt qu'une tabulation, pour l'OutputFormat `TextOutputFormat`.

Les OutputFormats Hadoop

- **Enfin, il est évidemment possible de créer ses propres OutputFormats, tout comme pour les InputFormats.**

- **Cela se fait de manière similaire:**

  - **En créant une classe héritant de la classe `FileOutputFormat` (ou `DBOutputFormat`, etc.), paramétrisée avec un type de clef et un type de liste de valeurs (par exemple `List<Text>`).**

  - **En créant une classe héritant de la classe `RecordWriter`, paramétrisée de la même manière. Il s'agit du pendant de la classe `RecordReader` vue précédemment, pour les OutputFormats.**

Les types Writable spécifiques

- **Jusqu'ici, on a toujours utilisé des types « simples » pour les clefs et valeurs utilisées dans les programmes d'exemple: par exemple `Text`, ou encore `IntWritable`.**

- **Hadoop permet également de définir ses propres types spécifiques, pour la clef ou pour la valeur. On pourrait ainsi avoir un objet passé en temps que valeur d'un programme map/reduce.**

- **Il suffit pour ce faire de créer une nouvelle classe implémentant une interface Hadoop.**

- **Les types vus précédemment: `IntWritable`, `Text`, `LongWritable`, etc. sont justement des types Hadoop implémentant cette interface autour des types Java standards (Int, String, Long, etc.).**

Les types Writable spécifiques

- **Pour créer un type Writable spécifique, on doit implémenter l'interface Hadoop `WritableComparable`.**

- **Il faut, au sein de la classe, implémenter au minimum les fonctions suivantes:**

  - **`write`**
    Pour écrire les données de la classe sur HDFS (et en interne pour Hadoop).

  - **`readFields`**
    Pour lire les données de la classe depuis HDFS (idem)

Les types Writable spécifiques

- `compareTo`

  Pour comparer l'objet courant à une autre instance du même type; la fonction doit renvoyer -1, 0 ou 1 si l'objet a une « valeur » respectivement inférieure, égale ou supérieure à la « valeur » de l'objet passé en argument. Cette méthode est avant tout nécessaire pour les clefs; elle a peu d'importances pour les types utilisés comme des valeurs.

- **hashcode**

  Doit générer un *hash* décrivant l'objet; ce *hash* permet à Hadoop d'identifier à quels reduceurs il doit envoyer quels couples (clef;valeur) lors du *shuffle*, entre l'opération map et reduce. Ce *hash* doit être identique pour une « valeur » donnée d'un objet; et il ne doit pas varier entre deux exécutions du programmes (ou d'une machine virtuelle Java à une autre). Cela signifie qu'utiliser la fonction Java standard *hashcode* sur l'objet ne fonctionnera possiblement *pas*; il est préférable d'implémenter sa propre fonction. Là aussi, cette méthode est nécessaire pour les clefs.

Les types Writable spécifiques

- **A noter que si on souhaite créer un type spécifique uniquement pour les valeurs, on peut implémenter à la place l'interface Writable; qui ne nécessitera pas d'implémentation des méthodes `compareTo` et `hashcode`.**

- **Ce type ne sera alors cependant pas utilisable en tant que clef dans un programme map/reduce Hadoop.**

- **A moins que cela n'ait pas de sens, on considère généralement qu'il est plus « propre » de créer un type spécifique de telle sorte qu'il soit à la fois « writable » et « comparable », et puisse donc être utiliser comme une valeur ou comme une clef.**

Les types Writable spécifiques - Exemple

- **On souhaite créer un type spécifique pour stocker nos valeurs dans l'exemple du réseau social / des « amis en commun ».**

- **La classe s'appellera « `FriendsListWritable` » et stockera la liste des amis liés à un utilisateur du réseau.**

- **On va donc implémenter notre propre type Writable.**

- **Comme on souhaite s'en servir uniquement comme valeur, les méthodes `compareTo()` et `hashcode()` ont moins d'importance.**

Les types Writable spécifiques - Exemple

- **Le code:**

```
package org.mbds.hadoop.friends;

import …

public class FriendsListWritable implements
WritableComparable<FriendsListWritable> {
    private ArrayList<String> friends=new ArrayList<String>();

    public void write(DataOutput out) throws IOException {
        Iterator<String> iterator=friends.iterator();
        String line="";
        while(iterator.hasNext()){
                String element=iterator.next();
                if(!line.equals(""))
                    line+=",";
                line+=element;
```

Les types Writable spécifiques - Exemple


```
            }
            out.writeChars(line);
    }

    public void readFields(DataInput in) throws IOException {
        String line=in.readLine();
        String[] elems=line.split(",");
        friends=new ArrayList<String>(Arrays.asList(elems));
    }

      public int getsize()
    {
        return(friends.size());
    }

    public int compareTo(FriendsListWritable o) {
```

Les types Writable spécifiques - Exemple


```
        int mysize=friends.size();
        int theirsize=o.getsize();
        return (mysize < theirsize ? -1 : (mysize==theirsize ? 0 : 1));
    }

    public int hashCode() {
        return friends.hashCode();
    }
}
```