Rapport TP2

Willian Ver Valen Paiva Alan Guitard

October 8, 2016

Contents

1 Introduction

Le but de ce TP est d'écrire un `InputFormat` personnalisé pour le framework Mapper/Reducer proposé par Hadoop.

L'objet InputFormat La documentation d'Hadoop nous renseigne sur cet objet en nous expliquant son utilisation:

1. Il valide la spécification d'entrée du job.
2. Il sépare les fichiers d'entrée avec une logique qu'on aura créé par le biais des `InputSplits` (Chaque "splits" sera traité par un mapper).
3. Il propose l'implémentation d'un lecteur de fichier d'entrée destiné à être utilisé. Ce lecteur lit les fichiers dont les données ont été divisés en "splits" et les rend accessible au mapper.

2 Exercice 1: Point2DWritable

L'objectif de se t'aider est d'écrire un objet implémentant l'interface `Writable` qui représentera un point décrit par des coordonnées X et Y, afin que les Mapper/Reducer puissent s'échanger cette objet. En effet, les objets lus ou écrit par ce job ont besoin d'implémenter les fonctions
`public void readFields(DataInout in) throws IOException` et
`public void write(DataOutput out) throws IOException`.

Pour coder le point en lui-même, l'objet `Point2D.Double`.

3 Exercice 2: RandomPointInputFormat

Cet objet doit proposer deux méthodes, implémentant l'objet abstrait `InputFormat<K,T>`. Après que les types génériques soient changés en `IntWritable` pour le K et `Point2DWritable` pour le T, ces méthode deviennent les suivantes:

```
@Override
        public RecordReader<IntWritable, Point2DWritable>
            createRecordReader(InputSplit arg0, TaskAttemptContext
            arg1)
                        throws IOException, InterruptedException {
                return new RandomPointReader();
        }


@Override
        public List<InputSplit> getSplits(JobContext arg0) throws
            IOException, InterruptedException {
        int splits = Integer.parseInt(arg0.getConfiguration().get("
            splits"));
```

```
        List<InputSplit> list = new ArrayList<InputSplit>()
            ;
        for(int i=0;i< splits;i++)
                list.add(new FakeInputSplit());
        return list;
    }
```

3.1 FakeInputSplit

Cette classe implémente l'interface `InputSplit` et est une classe qui simulera un bloc de données. En effet, dans ce TP, nous n'allons pas lire les points en deux dimensions dans des fichiers mais ils seront générés directement par l'objet qui suit.

3.2 RandomPointReader

Une classe implémentant l'interface `RecordReader` doit lire un à un, en itérant grâce à la fonction `nextKeyValue()` pour changer de couple clé/valeur, et proposer de récupérer ce couple courant par des getters. Comme nous simulons la lecture des points en deux dimensions, `nextKeyValue()` génère simplement un point aléatoire et donc le rend accessible par les getters.

4 Exercice 3: Test du RandomPointInputFormat

Pour tester ces implémentations, la configuration du job doit être changé dans la fonction main:

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.set("splits",args[1]);
    conf.set("points",args[2]);
    Job job = Job.getInstance(conf, "TP3");
    job.setNumReduceTasks(1);
    job.setJarByClass(TP3.class);
    job.setMapperClass(TP3Mapper.class);
    job.setMapOutputKeyClass(IntWritable.class);
    job.setMapOutputValueClass(Point2DWritable.class);
    job.setReducerClass(TP3Reducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(Point2DWritable.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setInputFormatClass(RandomPointInputFormat.class);
    FileOutputFormat.setOutputPath(job, new Path(args[0]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

Nous pouvons voir ici que les classes OutPut ont été changés, la clé est devenu un `IntWritable`, non plus un `Text`, et les valeurs des `Point2DWritable` et non plus des `IntWritable`.

Nous avons aussi ajouté la possibilité de configurer le mapping/reducing par les trois premières lignes, qui permettent de paramétrer les nombres d
e `FakeInputSplit` généré et le nombre de points générés contenus dans ces "splits".

5 Exercice 4: Calcul de PI

Cet exercice doit nous entraîner à une utilisation de notre programme en nous faisant calculer une approximation de la valeur de PI par l'algorithme de Monte-Carlo géométrique.

Explications Quand nous générons des points aléatoires entre [0,0] et [1,1],la probabilité que ces points soient dans le quart de cercle (de centre [0,0]) est de $\pi/4$.

5.1 Le Mapper

```
public void map(IntWritable key, Point2DWritable value, Context
    context
        ) throws IOException, InterruptedException {
        double x = value.getPoint().getX();
        double y = value.getPoint().getY();
        if(x*x+y*y <= 1){
            context.write(new Text("inside"), new IntWritable(1));
        }
    }
```

Le mapper est chargé de déterminer si le point qu'il reçoit est dans le quart de cercle, qui l'est si x*x+y*y est inférieure ou égale à 1. Si c'est le cas, il l'écrit dans le contexte et l'envoie au reducer.

5.2 Le Reducer

```
public void reduce(Text key, Iterable<IntWritable> values,
                    Context context
                    ) throws IOException, InterruptedException {
        int split = Integer.parseInt(context.getConfiguration().get("
            splits"));
        int points = Integer.parseInt(context.getConfiguration().get("
            points"));
        double success = 0;
        double drops = (split*points);
        for(IntWritable value : values){
            success += value.get();
        }
        context.write(new Text("PI"), new DoubleWritable(4*(success/
            drops)));
    }
```

Sachant que l'on as $\pi/4$ chances d'avoir un point dans le quart de cercle, nous pouvons calculer notre ratio réel de points à l'intérieur. En multipliant ce ratio par 4, nous avons donc une approximation de π.