Projet Big Data

Réalisé par Yassine Laghrabli

Année 2023/2024

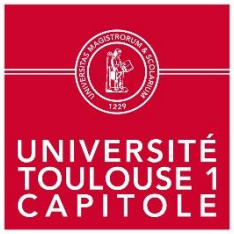

Table des matières

Introduction

Ce projet vise à nous familiariser avec la manipulation de grands volumes de données stockées dans HDFS. Dans la première partie, nous devions télécharger des données concernant les salaires des employés municipaux de la ville de New York. En utilisant Scala, Pig et MapReduce, nous avions pour tâche de nettoyer ces données, de calculer le salaire horaire des différents employés et de leur attribuer un sexe, grâce à un fichier associant noms et sexes. L'objectif est de structurer les données salariales afin d'analyser le salaire moyen annuel par sexe, par lieu de travail par sexe et par sexe.

Dans une seconde partie, nous devions importer des logs sur hdfs et extraire la date et l'heure ainsi que l'adresse IP avec des expressions régulières et compter le nombre de hit par jour par adresse et par heure par adresse.

Partie 1 : Pratique de Map/Reduce, Scala & PIG

I – MAP/REDUCE

A- Nettoyage des données

Pour la partie Nettoyage des données nous avons utilisé 2 mapper et 1 reducer

Class CsvDriver :

```
package csvClean;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;

public class CsvDriver {
    public static void main(String[] args) throws Exception {
          if (args.length != 3) {
               System.out.printf("Usage: CsvDriver <input csv fiscal> <input csv gender> <output dir>\n");
                 System.exit(-1);
               }
          Configuration conf = new Configuration();
          Job job = new Job(conf, "Reduce mappers");
          job.setJarByClass(CsvDriver.class);
          job.setReducerClass(CsvReducer.class);
          job.setOutputKeyClass(Text.class);
          job.setOutputValueClass(Text.class);
          MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class, CsvMapper1.class);
          MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class, CsvMapper2.class);
          FileOutputFormat.setOutputPath(job, new Path(args[2]));
          boolean success = job.waitForCompletion(true);
          System.exit(success ? 0 : 1);
    }
}
```

Dans la classe CsvDriver, nous prenons deux fichier en entrés, le fichier csv fiscal qui contient toutes les données lié au salaires des agents (5 millions de ligne) et le fichier csv associant noms et sexe.

Ce Driver prend deux mapper en argument et renvoi des clés, valeurs de type Text. Le premier argument étant le fichier fiscal, le deuxième argument et le fichier lié aux nom et le troisième argument est le nom des fichiers ou seront stockés les clés valeurs de sortie de ce traitement.

Class CsvMapper1 :

```
package csvClean;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class CsvMapper1 extends Mapper<LongWritable, Text, Text, Text> {
  public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
            String line = value.toString();
            String[] row = value.toString().split(",");
            if (line == null || line.isEmpty() || row.length != 17){return;}
            if ((row[4] == "")||(row[4]) == null || (row[4].isEmpty())){return;}
            if (row[0].matches("[0-9]+")) {
                float hours = Float.valueOf(row[12]);
                if (hours < 0){
                    hours = hours * (-1);
                }
                float salary = Float.valueOf(row[13]);
                if (salary < 0){
                    salary = salary * (-1);
                }
                float baseSalary = Float.valueOf(row[10]);
                if (baseSalary < 0){
                    baseSalary = baseSalary * (-1);
                }
                if ((hours != 0) && (salary != 0)) {
                    String salaryPerHour = Float.toString(salary / hours);
                     String rowMaj = value.toString() + "," + salaryPerHour;
                     context.write(new Text(row[4].toLowerCase()), new Text(rowMaj));
                } else if (baseSalary != 0){
                     String payBasis = row[11];
                     if (payBasis.equals("per Hour")) {
                        String rowMaj = value.toString() + "," + row[10];
                        context.write(new Text(row[4].toLowerCase()), new Text(rowMaj));
                     }
                     if (payBasis.equals("per Day")) {
                        float hoursPerDay = 7.5f;
                        String rowMaj = value.toString() + "," + Float.toString(baseSalary / hoursPerDay);
                        context.write(new Text(row[4].toLowerCase()), new Text(rowMaj));
                     }
                     if (payBasis.equals("per Annum") || payBasis.equals("Prorated Annual")) {
                        int hoursPerYear = 1950;
                        String rowMaj = value.toString() + "," + Float.toString(baseSalary / hoursPerYear);
                        context.write(new Text(row[4].toLowerCase()), new Text(rowMaj));
}}}}}
```

Dans la classe CsvMapper1, nous traitons le fichier fiscal. Nous avons initialement constaté que certaines lignes du fichier présentent des colonnes manquantes, ce qui pose des problèmes pour la conversion des salaires en Float. Pour remédier à cela, nous excluons les lignes dont la longueur diffère de 17, celles qui sont entièrement vides et celles qui ne contiennent pas de nom dans la colonne dédiée, car l'analyse se base sur le sexe et l'absence de nom rend impossible l'association de la ligne à un sexe.

Nous avons également observé que dans certaines lignes, la base salariale, le nombre d'heures travaillées ainsi que la rémunération réelle sont soit nulles, soit égales à zéro, soit négatives.

Hypothèse 1 : Si le nombre d'heures travaillées, le salaire perçu et la base salariale sont égaux à zéro, nous décidons d'ignorer la ligne.

Hypothèse 2 : Si le nombre d'heures travaillées et le salaire perçu sont différents de zéro, nous calculons le salaire par heure en divisant le salaire perçu par le nombre d'heures travaillées.

Si le nombre d'heures travaillées et le salaire perçu sont égaux à zéro, mais que la base salariale est différente de zéro alors,

Hypothèse 3 : si la base salariale est par heure, nous la conservons telle quelle.

Hypothèse 4 : si la base salariale est par jour, nous la divisons par 7,5.

Hypothèse 5 : si la base salariale est annuelle, nous la divisons par 1950.

Une fois les calculs effectués, nous stockons les clés et les valeurs dans le contexte pour qu'elles soient traitées dans le reducer. La clé correspond au nom de l'employé et la valeur est la ligne du CSV avec la colonne salaire ajoutée.

Class CsvMapper2 :

```
package csvClean;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class CsvMapper2 extends Mapper<LongWritable, Text, Text, Text> {
  public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
            String[] row = value.toString().split(",");
            context.write(new Text(row[0].toLowerCase()), new Text(row[1]+","+row[2]));
  }
}
```

Dans la classe CsvMapper2, nous traitons le fichier contenant les noms et les sexes. Nous envoyons vers le reducer des paires clé-valeur de type Text, où la clé est le nom et la valeur est le sexe. L'objectif est d'associer le sexe à chaque ligne du CSV qui possède un nom correspondant dans le fichier des noms.

Class CsvReducer :

```
package csvClean;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Reducer;


public class CsvReducer extends Reducer<Text, Text, Text, NullWritable> {
  public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
            List<String> rows = new ArrayList<>();
            String sexe = "";
            double probability = 0.0;
            for (Text value : values) {
                String row = value.toString();
```

```
                if (row.matches("[MF],[\\d].*")){
                    sexe += row.split(",")[0];
                    probability += Double.valueOf(row.split(",")[1]);
                } else if (!row.isEmpty()) {
                    rows.add(row);
                }
            }
            if (sexe.equals("")) {
                sexe = "N";
                for (String row : rows) {
                    context.write(new Text(row + "," + sexe), NullWritable.get());
                }
            } else {
                Random random = new Random();
                for (String row : rows) {
                    double randomP = random.nextDouble();
                    if (randomP <= probability) {
                        context.write(new Text(row + "," + sexe), NullWritable.get());
                    } else {
                        if (sexe.equals("F")) {
                            context.write(new Text(row + ",M"), NullWritable.get());
                        } else {
                            context.write(new Text(row + ",F"), NullWritable.get());
}}}}}}
```

La classe CsvReducer traite les entrées de paires clé-valeur de type Text et génère en sortie des paires où la clé est également de type Text et la valeur est de type NullWritable, cette dernière indiquant l'absence de valeur associée. La clé en sortie est une ligne de fichier CSV mise à jour avec l'informations supplémentaires sur le sexe de la personne mentionnée.

Avant de traiter les données, la classe prépare une liste List<String> pour stocker toutes les lignes de données CSV pertinentes et une variable de type String pour accumuler l'information de sexe, si disponible. Cette préparation est nécessaire car la variable values, contenant les données et les sexes associés à un même nom, ne peut être parcourue qu'une seule fois.

La classe parcourt chaque élément de values avec une boucle for. Si un sexe est identifié (représenté par une chaîne contenant "M" ou "F" et une probabilité), cette information est stockée. Les lignes CSV sans indication de sexe sont également conservées pour un traitement ultérieur.

Si, après le parcours de values, aucun sexe n'a été déterminé pour un nom (par exemple, si le nom n'existe pas dans le fichier des genres), la classe attribue "N" comme sexe par défaut pour toutes les lignes concernées.

Pour les noms associés à des probabilités (par exemple, 0.6 pour une femme, impliquant 40 % de chance que le nom soit masculin), le reducer utilise un objet Random pour générer un nombre aléatoire entre 0 et 1. Si ce nombre est inférieur ou égal à la probabilité associée au sexe, le sexe défini est conservé pour la ligne en question ; sinon, le sexe opposé est attribué.

Enfin, la classe concatène l'information de sexe à chaque ligne CSV stockée, créant ainsi une nouvelle colonne. Ces lignes mises à jour sont ensuite écrites dans le contexte de sortie, qui sera utilisé pour générer le fichier final avec les données nettoyées.

Commande d'exécution sous cloudera:

```
[cloudera@quickstart ~]$ cd /home/cloudera/workspace/training/bin
[cloudera@quickstart bin]$ jar cvf csv.jar csvClean
added manifest
adding: csvClean/(in = 0) (out= 0)(stored 0%)
adding: csvClean/CsvMapper2.class(in = 2203) (out= 806)(deflated 63%)
```

```
adding: csvClean/CsvMapper1.class(in = 3457) (out= 1524)(deflated 55%)
adding: csvClean/CsvReducer.class(in = 3788) (out= 1586)(deflated 58%)
adding: csvClean/CsvDriver.class(in = 1982) (out= 996)(deflated 49%)
[cloudera@quickstart bin]$ hadoop jar csv.jar csvClean.CsvDriver fiscal gender cleanData
```

Extrait du Résultat :

```
2018,747,DEPT OF ED PER SESSION TEACHER,RANDAZZO,ADELE,,09/08/2015,MANHATTAN,TEACHER- PER SESSION,ACTIVE,33.18,per Day,0,1079.5
0,0,0.00,4.75,4.4240003,F
2018,747,DEPT OF ED PER SESSION TEACHER,CARAMBIA,ADELE,,07/30/1996,MANHATTAN,TEACHER- PER SESSION,ACTIVE,33.18,per Day,0,2612.8
8,0,0.00,0.00,4.4240003,F
2018,466,COMMUNITY COLLEGE (MANHATTAN),SHTERN,ADELE,,09/11/1996,MANHATTAN,NON-TEACHING ADJUNCT II,CEASED,42.66,per Hour,0,100.0
0,0,0.00,0.00,42.66,F
2018,742,DEPT OF ED PEDAGOGICAL,PATURZO,ADELE,L,02/05/2007,MANHATTAN,SCHOOL SECRETARY,ACTIVE,63281.00,per Annum,0,60042.23,0,0.0
0,0.00,32.451794,F
2018,300,BOARD OF ELECTION POLL WORKERS,HARRIGAN,ADELE,,09/01/2010,MANHATTAN,ELECTION WORKER,ACTIVE,1.00,per Hour,0,300.00,0,0.0
0,0.00,1.00,F
2018,747,DEPT OF ED PER SESSION TEACHER,TUOMI,ADELE,G,02/01/1985,MANHATTAN,TEACHER- PER SESSION,ACTIVE,33.18,per Day,0,1618.05,
0,0.00,0.00,4.4240003,F
2018,742,DEPT OF ED PEDAGOGICAL,DURICK,ADELE,H,09/06/1978,OTHER,TEACHER SPECIAL EDUCATION,CEASED,100249.00,per Annum,0,600.00,0,
0.00,0.00,51.409744,F
2018,820,ADMIN TRIALS AND HEARINGS,COHEN,ADELE,H,07/04/2011,BROOKLYN,HEARING OFFICER,ACTIVE,53.85,per Hour,846.25,44984.38,0,0.0
0,0.00,53.15732,F
2018,742,DEPT OF ED PEDAGOGICAL,LONG,ADELE,A,09/02/1997,MANHATTAN,TEACHER,ACTIVE,113685.00,per Annum,0,108742.84,0,0.00,0.00,58.
3,F
2018,747,DEPT OF ED PER SESSION TEACHER,BASILE-DIRENZO,ADELE,D,08/30/2007,MANHATTAN,TEACHER- PER SESSION,ACTIVE,33.18,per Day,0,
357.98,0,0.00,0.00,4.4240003,F
```

B- Analyse

Pour la partie Analyse, nous avons utilisé un mapper et un reducer.

Class FaDriver :

```java
package fiscalAnalysis;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FaDriver {
    public static void main(String[] args) throws Exception {
         if (args.length != 2) {
                System.out.printf("Usage: LogDriver <input dir> <output dir>\n");
                System.exit(-1);
           }
           Job job = new Job();
           job.setJarByClass(FaDriver.class);
           job.getConfiguration();
           job.setJobName("Fiscal Analysis");
           FileInputFormat.setInputPaths(job, new Path(args[0]));
           FileOutputFormat.setOutputPath(job, new Path(args[1]));
           job.setMapperClass(FaMapper.class);
           job.setReducerClass(FaReducer.class);
           job.setMapOutputKeyClass(Text.class);
           job.setMapOutputValueClass(Text.class);
           job.setOutputKeyClass(Text.class);
           job.setOutputValueClass(Text.class);
           boolean success = job.waitForCompletion(true);
```

```
            System.exit(success ? 0 : 1);
    }
}
```

La classe FaDriver prend deux argument, le fichier txt contenant les lignes filtrés et mise à jour avec le salaire et le sexe et en deuxième argument le nom du nouveau fichier ou seront stocké les données calculé à la fin de ce nouveau traitement.

Class FaMapper :

```
package fiscalAnalysis;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class FaMapper extends Mapper<LongWritable, Text, Text, Text> {
  public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException {
    String[] row = value.toString().split(",");
    if (row.length != 19){ return; }
    context.write(new Text("rows"), new Text(row[0].trim() + "," + row[7].trim() + "," + row[18].trim() +
"," + row[17]));
            //rows : date,work loc,sexe,pay per hour
  }
}
```

Le mapper prend en entré des valeurs de type Text et retourne une paire clés valeurs de type Text chacune. Ce mapper récupère chacune des lignes du fichier traité précédemment.

Le mapper vérifie si la valeur récupérer contient exactement 19 éléments. Cette vérification s'assure que seules les lignes complètes sont traitées, excluant ainsi toute ligne mal formée ou incomplète.

Si la ligne est conforme, la méthode extrait certains éléments spécifiques de cette ligne :

- `row[0].trim()` : la date, avec suppression des espaces de début et de fin.
- `row[7].trim()` : le lieu de travail, également nettoyé des espaces superflus.
- `row[18].trim()` : le sexe, nettoyé des espaces.
- `row[17]` : la rémunération par heure, sans nettoyage supplémentaire pour les espaces dans cet exemple.

Ces éléments sont ensuite concaténés, séparés par des virgules, et écrits dans le contexte de sortie avec la clé "rows". Ainsi, chaque valeur de sortie contient une concaténation de la date, du lieu de travail, du sexe et de la rémunération par heure.

Class FaReducer :

```
package fiscalAnalysis;

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class FaReducer extends Reducer<Text, Text, Text, Text> {
```

```
  public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
        HashMap<String, Float> analysisByDate = new HashMap<>();
        HashMap<String, Float> analysisBySexe = new HashMap<>();
        HashMap<String, Float> analysisByWorkLoc = new HashMap<>();
        HashMap<String, Integer> countByDate = new HashMap<>();
        HashMap<String, Integer> countBySexe = new HashMap<>();
        HashMap<String, Integer> countByWorkLoc = new HashMap<>();
        List<String> dates = new ArrayList<String>();
        List<String> workLocs = new ArrayList<String>();
        List<String> sexes = new ArrayList<String>();
        for (Text value : values) {
            String[] row = value.toString().split(",");
            Float payPerHour = Float.valueOf(row[3]);
            if ((row[0] != null) && (!row[0].trim().isEmpty()) && (row[2].matches("[MF]"))){
                String dateSexe = row[0] + " " + row[2];
                 if (!analysisByDate.containsKey(dateSexe)) {
                     analysisByDate.put(dateSexe, 0.0f);
                     countByDate.put(dateSexe, 0);
                      dates.add(dateSexe);
                 }
                 analysisByDate.put(dateSexe, analysisByDate.get(dateSexe) + payPerHour);
                 countByDate.put(dateSexe, countByDate.get(dateSexe) + 1);
           }
           if ((row[1] != null) && (!row[1].matches("\\s*")) && (row[2].matches("[MF]"))){
               String locationSexe = row[1] + " " + row[2];
               if (!analysisByWorkLoc.containsKey(locationSexe)) {
                     analysisByWorkLoc.put(locationSexe, 0.0f);
                     countByWorkLoc.put(locationSexe, 0);
                     workLocs.add(locationSexe);
               }
               analysisByWorkLoc.put(locationSexe, analysisByWorkLoc.get(locationSexe) + payPerHour);
               countByWorkLoc.put(locationSexe, countByWorkLoc.get(locationSexe) + 1);
            }
            if (row[2].matches("[MF]")){
                 if (!analysisBySexe.containsKey(row[2])) {
                     analysisBySexe.put(row[2], 0.0f);
                     countBySexe.put(row[2], 0);
                     sexes.add(row[2]);
                  }
                  analysisBySexe.put(row[2], analysisBySexe.get(row[2]) + payPerHour);
                   countBySexe.put(row[2], countBySexe.get(row[2]) + 1);
            }
         }
          context.write(new Text("Analyse salaire moyen par annee : "), new Text(""));
          for (String date : dates){
          context.write(new Text(date), new Text(Float.toString(analysisByDate.get(date) /
countByDate.get(date))));
          }
          context.write(new Text("Analyse salaire moyen par Sexe : "), new Text(""));
          for (String sexe : sexes) {
              context.write(new Text(sexe), new Text(Float.toString(analysisBySexe.get(sexe) /
countBySexe.get(sexe))));
          }
          context.write(new Text("Analyse salaire moyen par Lieu de travail : "), new Text(""));
          for (String workLoc : workLocs) {
              context.write(new Text(workLoc), new Text(Float.toString(analysisByWorkLoc.get(workLoc) /
countByWorkLoc.get(workLoc))));
}}}
```

La classe FaReducer initialise plusieurs HashMap pour stocker les analyses des salaires moyens (analysisByDate, analysisBySexe, analysisByWorkLoc) et les comptages (countByDate, countBySexe, countByWorkLoc). Elle utilise également des listes (dates, workLocs, sexes) pour enregistrer les clés uniques rencontrées pendant le traitement.

Traitement des valeurs :

- Sépare la valeur en ses composantes (date, lieu de travail, sexe, et salaire par heure).
- Vérifie et traite les données pour chaque type d'analyse (par date et sexe, par lieu de travail et sexe, et par sexe uniquement).
- Met à jour les sommes des salaires et les comptages pour chaque catégorie pertinente.
- Calcul des moyennes : Après avoir traité toutes les valeurs, la classe calcule les salaires moyens pour chaque catégorie en divisant la somme totale du salaire par heure par le nombre de comptages correspondants.

Finalement, la classe écrit dans le contexte de sortie les résultats des analyses avec le Type Text :

Pour chaque (année, lieu de travail) par sexe et sexe, le salaire moyen est calculé et écrit.

Commande d'exécution sous cloudera:

```
[cloudera@quickstart ~]$ cd /home/cloudera/workspace/training/bin
[cloudera@quickstart bin]$ jar cvf fa.jar fiscalAnalysis
added manifest
adding: fiscalAnalysis/(in = 0) (out= 0)(stored 0%)
adding: fiscalAnalysis/FaDriver.class(in = 1890) (out= 998)(deflated 47%)
adding: fiscalAnalysis/FaReducer.class(in = 5041) (out= 2084)(deflated 58%)
adding: fiscalAnalysis/FaMapper.class(in = 2293) (out= 858)(deflated 62%)
[cloudera@quickstart bin]$ hadoop jar fa.jar fiscalAnalysis.FaDriver newFiscal fiscalAnalysisResult
```

Extrait du Résultat :

```
Analyse salaire moyen par annee :
2023 F  44.18155
2022 M  532.695
2023 M  548.6898
2022 F  41.86914
2019 F  38.954395
2017 F  38.77973
2018 F  41.489254
2015 F  24.768625
2014 F  24.588583
2016 F  27.193129
2021 F  45.29742
2020 F  43.75925
2019 M  471.65598
2021 M  546.42584
2020 M  555.2408
2016 M  34.6261
2018 M  508.4958
2017 M  453.5384
2014 M  31.580608
2015 M  31.643705
Analyse salaire moyen par Sexe :
F       37.219883
M       368.13953
```

```
Analyse salaire moyen par Lieu de travail :
MANHATTAN F     37.6459
MANHATTAN M     665.167
BRONX F 37.68252
BROOKLYN F      34.92097
QUEENS F        43.488632
QUEENS M        43.270214
BRONX M 40.288353
BROOKLYN M      40.7372
OTHER F 57.676804
Bronx F 33.04176
RICHMOND F      30.884731
OTHER M 59.28866
Manhattan F     36.457066
RICHMOND M      38.01759
WESTCHESTER F   36.98702
Queens F        37.45161
Queens M        43.65518
NASSAU M        54.271046
SULLIVAN M      39.840427
WESTCHESTER M   40.275986
ULSTER M        41.688198
Manhattan M     37.70441
Bronx M 36.878918
ALBANY M        48.212994
Richmond F      36.536983
ULSTER F        39.75729
```

II- SCALA

A- Nettoyage des données

L'objectif est maintenant de réaliser des analyses similaires en Scala. Pour cela, nous commençons par importer deux fichiers CSV en tant que variables, via la fonction `textFile` de Spark.

```
val fiscal = sc.textFile("/user/cloudera/fiscal")
val gender = sc.textFile("/user/cloudera/gender")
```

Étant donné que ces fichiers sont au format CSV, il est nécessaire d'éliminer la première ligne contenant les en-têtes. Pour y parvenir, nous déclarons deux nouvelles variables pour stocker ces premières lignes.

```
val fiscalHeader = fiscal.first()
val genderHeader = gender.first()
```

Nous créons ensuite une variable fiscalData qui contiendra les données filtrées du fichier fiscal, qui contiennent l'ajout du salaire horaire. Ce processus se décompose en plusieurs étapes : filtrage pour éliminer l'en-tête, séparation des ligne par la virgule, filtrage selon le nombre d'éléments par ligne, et enfin, transformation pour calculer le salaire horaire.

```
val fiscalData = fiscal.filter(_ != fiscalHeader).
map(_.split(",")).filter(_.length == 17).
map(row => (row, row(12).toFloat, row(13).toFloat, row(11), row(10).toFloat)).
map{ case (row, hoursRaw, salaryRaw, payBasis, baseSalaryRaw) =>
  val hours = if (hoursRaw < 0) -hoursRaw else hoursRaw
  val salary = if (salaryRaw < 0) -salaryRaw else salaryRaw
  val baseSalary = if (baseSalaryRaw < 0) -baseSalaryRaw else baseSalaryRaw
  (row, hours, salary, payBasis, baseSalary)
}.
map{ case (row, hours, salary, payBasis, baseSalary) =>
  val rowMaj = if ((hours != 0) && (salary != 0)) {
    row(0) + "," + row(7).toLowerCase + "," + (salary / hours).toString
  } else if (hours == 0 && salary != 0 && baseSalary != 0) {
    payBasis match {
      case "Per Hour" => row(0) + "," + row(7).toLowerCase + "," + baseSalary.toString
      case "Per Day" => row(0) +  "," + row(7).toLowerCase + "," + (baseSalary/7.5).toString
      case "Per Annum" | "Prorated Annual" => row(0) + "," + row(7).toLowerCase + "," +
(baseSalary/1945).toString
      case _ => None
    }
  } else {
    None
  }
  (row(4).toLowerCase, rowMaj.toString)
}.filter(_._2 != "None")
```

Explication en détail des étapes de la variable fiscalData :

Filtrage pour éliminer l'entête :

```
filter(_ != fiscalHeader)
```

Séparation du contenu de chaque ligne par la virgule

```
map(_.split(","))
```

Filtrage pour éliminer les lignes qui n'ont pas 17 arguments

```
filter(_.length == 17)
```

Sélection des données utiles pour les calculs et analyses future :

```
map(row => (row, row(12).toFloat, row(13).toFloat, row(11), row(10).toFloat))
```

Ici nous avons mapper la ligne entière, la base salariale, le nombre d'heures travaillées ainsi que la rémunération réelle ainsi que base Salarial. Nous avons aussi converti en Float les valeurs numériques pour pouvoir effectuer les calculs

Conversion valeurs négatifs en valeurs positif :

```
map{ case (row, hoursRaw, salaryRaw, payBasis, baseSalaryRaw) =>
  val hours = if (hoursRaw < 0) -hoursRaw else hoursRaw
  val salary = if (salaryRaw < 0) -salaryRaw else salaryRaw
  val baseSalary = if (baseSalaryRaw < 0) -baseSalaryRaw else baseSalaryRaw
  (row, hours, salary, payBasis, baseSalary)
}
```

Pour chaque Valeur numérique, a savoir , la base salariale (baseSalary), le nombre d'heures travaillées (hours) ainsi que la rémunération réelle (salary) ont vérifie si les valeurs sont inférieur à 0 si la condition est vrai alors on ajoute (-) à la variable ce qui revient à multiplier par -1.

Calcul du salaire par heure :

```
map{ case (row, hours, salary, payBasis, baseSalary) =>
  val rowMaj = if ((hours != 0) && (salary != 0)) {
    row(0) + "," + row(7).toLowerCase + "," + (salary / hours).toString
  } else if (baseSalary != 0) {
    payBasis match {
      case "Per Hour" => row(0) + "," + row(7).toLowerCase + "," + baseSalary.toString
      case "Per Day" => row(0) +  "," + row(7).toLowerCase + "," + (baseSalary/7.5).toString
      case "Per Annum" | "Prorated Annual" => row(0) + "," + row(7).toLowerCase + "," +
(baseSalary/1945).toString
      case _ => None
    }
  } else {
    None
  }
  (row(4).toLowerCase, rowMaj.toString)
}
```

Pour calculer les salaires base heure pous avons procéder avec la même logique qu'en Map/Reduce avec les même hypothèse :

Hypothèse 1 : Si le nombre d'heures travaillées, le salaire perçu et la base salariale sont égaux à zéro, nous attribuons au salaire la valeur None.

Hypothèse 2 : Si le nombre d'heures travaillées et le salaire perçu sont différents de zéro, nous calculons le salaire par heure en divisant le salaire perçu par le nombre d'heures travaillées.

Si la base salariale est différente de zéro alors,

Hypothèse 3 : si la base salariale est par heure, nous la conservons telle quelle.

Hypothèse 4 : si la base salariale est par jour, nous la divisons par 7,5.

Hypothèse 5 : si la base salariale est annuelle, nous la divisons par 1950.

Après les calculs nous convertissons le salaire en string et nous concaténons toutes les valeurs utiles avec une virgule comme séparateur.

Suppression des lignes n'ayant pas remplis les conditions de calculs de salaire :

```
filter(_._2 != "None")
```

Cette variable retourne une paire contenant le nom de l'agent ainsi que la lignes concaténé contenant les valeurs sélectionnées à savoir l'année, le lieu de travail et le salaire horaire.

Maintenant que les données fiscal sont traité et nettoyé, il est temps de faire la même chose avec les données qui concerne les genres associés aux noms. Pour ce faire, nous allons crée la variable genderData dans laquelle nous allons supprimer l'entête et ensuite la mapper avec la paire nom et sexe associé au nom. Contrairement aux MapReduce, ici nous n'allons pas prendre en compte les probabilités pour attribuer le sexe à la personne.

```
val genderData = gender.filter(_ != genderHeader).
map(_.split(",")).filter(_.lengh == 3).
map row = {(row(0).toLowerCase, row(1))})
```

Maintenant que nous avons mapper similairement les variables fiscalData et genderData nous pouvons désormais les joindre pour associer chaque valeur de fiscal à un sexe.

```
val jointure = fiscalData.leftOuterJoin(genderData).
map{ case (name, (fisc, sexe)) => (name, (fisc, sexe.getOrElse("N")))}.
map{ case (name, (fisc, sexe)) => s"$name,$fisc,$sexe"}
```

Commande pour stocker le résultat :

```
jointure.saveAsTextFile("/user/cloudera/cleanData")
```

Extrait du Résultat :

```
rashawn,2023,queens,19.570408,M
rashawn,2023,brooklyn,16.482338,M
rashawn,2023,manhattan,17.812561,M
rashawn,2023,brooklyn,13.983075,M
rashawn,2023,bronx,13.871433,M
kwanza,2020,bronx,19.309126,F
kwanza,2020,manhattan,11.793169,F
kwanza,2021,bronx,22.57594,F
kwanza,2021,manhattan,15.372063,F
kwanza,2016,bronx,20.522478,F
kwanza,2016,bronx,22.17332,F
kwanza,2016,manhattan,11.771821,F
kwanza,2015,bronx,21.187414,F
kwanza,2015,bronx,20.264723,F
kwanza,2014,,20.291117,F
```

B- Analyse

Analyse par sexe :

Pour l'analyse par sexe, on crée une variable où l'on va récupérer le résultat de la jointure que l'on va ensuite diviser (split) et transformer en mappant les valeurs. On récupère le sexe et on convertit le salaire en Float pour ensuite stocker le résultat par tuple clé-valeurs, pour lequel la clé est le sexe et la valeur est un tuple contenant le salaire et la valeur 1 : `(sexe, (salaire, 1))`. La valeur 1 nous servira lorsqu'on fera le comptage pour calculer le salaire moyen par sexe. Après la transformation initiale, le flux de données est filtré pour ne retenir que les lignes correspondant aux sexes "M" ou "F". Les données sont ensuite agrégées par sexe en utilisant `.reduceByKey`. Cette opération combine les valeurs associées à chaque clé (sexe) en utilisant une fonction de réduction spécifiée. Enfin, la fonction `.mapValues` est appliquée pour transformer les valeurs agrégées (total du salaire et compteur) en salaire moyen.

```
val salaryPerSex = jointure.map(line => {
    val row = line.split(",")
    val sexe = row(4)
    val salary = row(3).toFloat
    (sexe, (salary, 1))
  }).
  filter{ case (sexe, _) => sexe == "M" || sexe == "F" }.
  reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2)).
  mapValues{ case (totalSalary, count) => totalSalary / count }
```

Résultat :

```
scala> salaryPerSexe.collect()foreach(println)
(F,39.972363)
(M,43.226345)
```

Analyse par année par sexe :

Le traitement pour analyser le salaire moyen par année et par sexe est identique à la méthode précédente (voir salaire moyen par sexe), le seul point qui change est la valeur de la clé sur laquelle on va agréger les données, car cette fois, on va concaténer l'année et le sexe.

```
val salaryPerYear = jointure.map(line => {
    val row = line.split(",")
    val year = row(1)
    val sexe = row(4)
    val salary = row(3).toFloat
    (year, sexe, (salary, 1))
}).
filter{ case (_, sexe, _) => sexe == "M" || sexe == "F" }.
map{ case (year, sexe, salaryCount) => (s"$year for $sexe", salaryCount) }.
reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2)).
mapValues{ case (totalSalary, count) => totalSalary / count}
```

Résultat :

```
scala> salaryPerYear.collect()foreach(println)
(2019 for M,43.191532)
(2023 for F,44.6028)
(2015 for F,35.652176)
(2020 for F,40.32192)
(2023 for M,46.47275)
(2021 for F,42.555386)
(2016 for F,37.467854)
(2020 for M,43.3197)
(2015 for M,39.450115)
(2021 for M,44.59895)
(2017 for F,41.711487)
(2022 for F,43.0544)
(2016 for M,41.623898)
(2017 for M,45.298122)
(2022 for M,46.139244)
(2018 for F,39.795044)
(2014 for F,34.512608)
(2014 for M,38.335106)
(2018 for M,43.860786)
(2019 for F,40.13503)
```

Analyse par lieu de travail par sexe :

Le traitement pour analyser le salaire moyen par année et par sexe est identique à la méthode précédente (voir salaire moyen par sexe), le seul point qui change est la valeur de la clé sur laquelle on va agréger les données, car cette fois, on va concaténer le lieu de travail et le sexe.

```
val salaryPerLoc = jointure.map(line => {
    val row = line.split(",")
    val loc = row(2)
    val sexe = row(4)
    val salary = row(3).toFloat
    (loc, sexe, (salary, 1))
}).
filter{ case (_, sexe, _) => sexe == "M" || sexe == "F" }.
filter{ case (loc, _, _) => loc != "" }.
map{ case (loc, sexe, salaryCount) => (s"$loc for $sexe", salaryCount) }.
reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2)).
mapValues{ case (totalSalary, count) => totalSalary / count}
```

Résultat :

```
scala> salaryPerLoc.collect()foreach(println)
(orange for M,42.063545)
(nassau for F,49.719555)
(sullivan for M,39.84623)
(ulster for F,39.611794)
(delaware for F,32.661762)
(brooklyn for M,41.061066)
(manhattan for M,46.899555)
(nassau for M,54.122635)
(schoharie for F,31.492365)
(ulster for M,42.025024)
(greene for F,29.784367)
(delaware for M,34.636696)
(other for F,47.646908)
(other for M,78.62474)
(washington dc for F,76.303185)
(greene for M,42.077175)
(schoharie for M,33.334667)
(putnam for F,38.01148)
(washington dc for M,58.40445)
(dutchess for M,47.849617)
(richmond for F,31.265884)
(putnam for M,36.26241)
(queens for F,44.195625)
(bronx for F,38.394566)
(albany for F,70.92442)
(richmond for M,38.30342)
(westchester for F,36.929234)
(sullivan for F,35.548416)
(bronx for M,40.806812)
(albany for M,48.007122)
(brooklyn for F,35.35088)
(orange for F,31.127586)
(manhattan for F,42.134193)
(queens for M,43.683975)
(westchester for M,40.462803)
```

III- PIG

L'objectif étant toujours d'effectuer les même analyse ainsi que le traitement de données effectuer précédemment en MapReduce et Scala.

A- Nettoyage des données

```
fiscalLoad = LOAD '/user/cloudera/fiscal' USING PigStorage(',') AS (year:chararray, f2:chararray,
f3:chararray, f4:chararray, name:chararray, f6:chararray, f7:chararray, location:chararray, f9:chararray,
f10:chararray, baseSalary:chararray, payBasis:chararray, hours:chararray, salary:chararray, f15:chararray,
f16:chararray, f17:chararray);

fiscalCleaned = FILTER fiscalLoad BY (hours MATCHES '[0-9.-]+') AND (year MATCHES '[0-9]+') and (salary
MATCHES '[0-9.-]+') and (payBasis MATCHES '[a-zA-Z ]+') and (baseSalary MATCHES '[0-9-.]+');

genderLoad = LOAD '/user/cloudera/gender' USING PigStorage(',') AS (name:chararray, sexe:chararray,
prob:chararray);

genderCleaned = Filter genderLoad BY (name != 'name');

genderMaj = FOREACH genderCleaned GENERATE LOWER(name) as name, sexe;

fiscalData = FOREACH fiscalCleaned GENERATE year, f2, f3, f4, name, f6, f7, location, f9, f10,
ABS((float)baseSalary) as baseSalary, payBasis, ABS((float)hours) as hours, ABS((float)salary) as salary,
f15, f16, f17;

fiscalMaj = FOREACH fiscalData GENERATE LOWER(name) as name, location, year,
```

```
  (hours > 0 and salary > 0 ? salary / hours :
   (baseSalary > 0 ?
    (payBasis == 'Per Hour' ? baseSalary :
     (payBasis == 'Per Day' ? baseSalary / 7.5f :
      (payBasis == 'Per Annum' OR payBasis == 'Prorated Annual' ? baseSalary / 1945.0f :
       0.0f))) :
    0.0f)) AS hourlyWage;

fiscalFinal = Filter fiscalMaj BY hourlyWage != 0

jointure = join fiscalFinal by name, genderMaj by name;
```

Ce script commence par importer les données des fichiers fiscal et gender stockés dans HDFS pour les stocker dans des variables (fiscalLoad et genderLoad) en utilisant PigStorage pour nommer et définir le type des colonnes. Les données fiscales sont filtrées pour éliminer les entrées invalides. Cela inclut la vérification que les heures, l'année, le salaire, la base de paye, et le salaire de base sont dans des formats attendus (numériques pour les montants et textuels pour la base de paye). Les noms dans l'ensemble de données de genre sont convertis en minuscules pour faciliter les jointures par la suite. Les données fiscales nettoyées subissent plusieurs transformations, notamment la conversion des montants salariaux en valeurs absolues et le changement des noms en minuscules. Pour le calcul du salaire, nous avons suivi la même logique que celle vue précédemment. Les données transformées sont encore filtrées pour ne garder que les entrées avec un salaire horaire non nul. Enfin, le script joint les données fiscales nettoyées et transformées avec les données de genre nettoyées sur le nom, permettant une analyse ultérieure basée sur le genre.

B- Analyse

Analyse par sexe :

Pour l'analyse par sexe, on regroupe les données jointes par sexe puis calculent le salaire horaire moyen pour chaque sexe. La première commande groupBySexe regroupe les données combinées (fiscales et de genre) par le champ sexe issu de l'ensemble de données de genre. La seconde commande avgWageBySexe parcourt chaque groupe créé et génère pour chaque sexe le salaire horaire moyen (avgHourlyWage)

```
groupBySexe = GROUP jointure BY genderMaj::sexe;

avgWageBySexe = FOREACH groupBySexe GENERATE group AS sexe, AVG(jointure.hourlyWage) AS avgHourlyWage;
```

Résultat :

```
F       39.96659665135547
M       43.21597047179531
```

Analyse par année par sexe :

Pour l'analyse par année par sexe, on va procéder comme pour l'analyse par sexe sauf que cette fois on va regrouper les données combinées par l'année et le sexe qui va créé une sorte de clé. A partir de la on va généré le salaire moyen par année par sexe.

```
groupByYearSexe = GROUP jointure BY (fiscalFinal::year, genderMaj::sexe);

avgWageByYearSexe = FOREACH groupByYearSexe GENERATE FLATTEN(group) AS (year, sexe),
AVG(jointure.hourlyWage) AS avgHourlyWage;
```

Résultat :

2014	F	34.51285514339365
2014	M	38.335608878234076
2015	F	35.65199658770605
2015	M	39.45047476762566
2016	F	37.46799813334899
2016	M	41.623660185905116
2017	F	41.71148246708861
2017	M	45.298033836818504
2018	F	39.79448656632636
2018	M	43.86040179612841
2019	F	40.135032855156034
2019	M	43.19212166841246
2020	F	40.294721340844916
2020	M	43.266486058815694
2021	F	42.55070233082229
2021	M	44.59368119276106
2022	F	43.05352921091
2022	M	46.139145164923214
2023	F	44.576968970773585
2023	M	46.46744433854039

Analyse par lieu de travail par sexe :

De la même manière, on va regrouper les données combinées le lieu de travail et le sexe comme clé. A partir de là on va générer le salaire moyen par lieu de travail par sexe.

```
groupByLocationSexe = GROUP jointure BY (fiscalFinal::location, genderMaj::sexe);

avgWageByLocationSexe = FOREACH groupByLocationSexe GENERATE FLATTEN(group) AS (location, sexe),
AVG(jointure.hourlyWage) AS avgHourlyWage;
```

Extrait du résultat :

```
BRONX   F       38.42154907440348
BRONX   M       40.81604839537227
Bronx   F       32.982009012392254
Bronx   M       37.071433922085596
OTHER   F       47.64691011833422
OTHER   M       77.21360206604004
ALBANY  F       70.92442249833492
ALBANY  M       48.00712704331907
GREENE  F       29.784366130828857
GREENE  M       42.07717139380319
NASSAU  F       49.71955705725628
NASSAU  M       54.122637598138105
ORANGE  F       31.127587685218224
ORANGE  M       42.06354761123657
PUTNAM  F       38.01147506350563
PUTNAM  M       36.26241425430967
QUEENS  F       44.20602482662322
QUEENS  M       43.6827307009258
Queens  F       37.29753637177992
Queens  M       44.014016761083035
ULSTER  F       39.611791953948384
ULSTER  M       42.02502374710338
BROOKLYN        F       35.3491555177992
BROOKLYN        M       41.060205509308766
DELAWARE        F       32.66176240306255
DELAWARE        M       34.63669451813639
DUTCHESS        M       47.84961732961066
RICHMOND        F       31.246221217016334
RICHMOND        M       38.27249575406692
```

Partie 2 : Utilisation des expressions régulières

I – MAP/REDUCE

Contrairement à l'exercice précèdent, ici on va utiliser un seul programme Map/Reduce pour faire directement les analyses Souhaiter.

Class LogDriver :

```java
package log;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class LogDriver {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.printf("Usage: LogDriver <input dir> <output dir>\n");
            System.exit(-1);
        }
        Job job = new Job();
        job.setJarByClass(LogDriver.class);
        job.getConfiguration();
        job.setJobName("Hit Count");
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(LogMapper.class);
```

```
        job.setReducerClass(LogReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        boolean success = job.waitForCompletion(true);
        System.exit(success ? 0 : 1);
    }
}
```

La classe LogDriver prend 2 arguments, un fichier de type txt en entré qui contient les différents hits et un fichier de sortie dans lequel seront stocké les différentes analyse qui seront effectuer à savoir les analyses des nombres de hits par heure et par jour des différentes adresses IP.

Class LogMapper :

```
package log;

import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class LogMapper extends Mapper<LongWritable, Text, Text, Text> {
  private final static Pattern p1 = Pattern.compile("(\\d+/\\w+/\\d+:\\d+)");
  private final static Pattern p2 = Pattern.compile("\\/twiki\\/bin\\/(\\w+)\\/");
  public void map(LongWritable key, Text value, Context context)
                  throws IOException, InterruptedException {
    String log = value.toString();
    Matcher m1 = p1.matcher(log);
    String ipA = "";
    String date = "";
    String result = "";
    if (m1.find()){
      date = m1.group(1);
    }
    Matcher m2 = p2.matcher(log);
    if (m2.find()){
      ipA = m2.group(1);
    }
    if (date != "" && ipA != ""){
      result += date + "," + ipA;
      context.write(new Text("hit"), new Text(result));
    }}}
```

La classe LogMapper reçoit en entrée un élément de type Text, représentant un enregistrement (hit). Cet enregistrement est converti en chaîne de caractères, puis analysé à l'aide de deux expressions régulières afin d'extraire la date au format 'JJ/MM/AAAA:HH' et l'adresse IP.

La première expression régulière utilisée pour extraire la date est "(\\d+/\\w+/\\d+:\\d+)"

La seconde, destinée à l'extraction de l'adresse IP, est "\\/twiki\\/bin\\/(\\w+)\\/".

La méthode .find est employée pour vérifier la présence de correspondances. Si au moins une des deux informations (date ou adresse IP) n'est pas trouvée, l'enregistrement est ignoré. Autrement, les deux informations extraites sont combinées dans une nouvelle variable, séparées par une virgule, pour former une chaîne de caractères.

Cette chaîne est ensuite stockée dans le context pour être transmise au reducer.

Class LogReducer :

```
package log;

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.util.HashMap;
import java.util.Map;

public class LogReducer extends Reducer<Text, Text, Text, Text> {
  public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
          HashMap<String, Integer> listeDay = new HashMap<>();
          HashMap<String, Integer> listeHour = new HashMap<>();
          for (Text value : values) {
            String[] v = value.toString().split("[:,]");
            if (v.length == 3) {
                    String day = v[0]+" "+v[2];
                    String hour = v[1]+" "+v[2];
        if (!listeHour.containsKey(hour)) {
          listeHour.put(hour, 0);
        }
        listeHour.put(hour, listeHour.get(hour)+1);
        if (!listeDay.containsKey(day)){
          listeDay.put(day, 0);
        }
        listeDay.put(day, listeDay.get(day) + 1);
            }
          }
          context.write(new Text("Nombre de hit par adresse par heure : "), new Text(""));
    for (Map.Entry<String, Integer> entry : listeHour.entrySet()) {
          String hourKey = entry.getKey();
          String hourValue = entry.getValue().toString();
          context.write(new Text(hourKey), new Text(hourValue));
    }
    context.write(new Text(""), new Text(""));
    context.write(new Text("Nombre de hit par adresse par jour : "), new Text(""));
          for (Map.Entry<String, Integer> entry : listeDay.entrySet()){
                    String dayKey = entry.getKey();
          String dayValue = entry.getValue().toString();
          context.write(new Text(dayKey), new Text(dayValue));
}}}
```

La classe LogReducer, prend en entré la liste de valeurs de type Text contenant la date et l'adresse des logs filtré précédemment. Deux structures de données sous forme de HashMpas sont utilisées pour stocker le nombre de hit par heure par adresse et par jour par adresse. La première HashMap, `listeHour`, comptabilise les hits par heure pour chaque adresse, tandis que la seconde, `listeDay`, fait de même mais sur une base quotidienne. Pour chaque valeur reçue, la méthode split est utilisé pour séparer la date, l'heure et l'adresse.

Si une clé (représentant une heure et adresse ou un jour et adresse) n'existe pas déjà dans le HashMap, elle est ajoutée avec une valeur initiale de 0, puis incrémentée pour chaque occurrence.

Pour finir on va parcourir chaque HashMap et stocké chaque paire clé, valeur dans le context. Ce qui correspond au nombre de hits par adresse par heure et au nombre de hits par adresse par jour.

II – SCALA

A- Nettoyage des données

```
val logs = sc.textFile("/user/cloudera/logss")

import scala.util.matching.Regex

val p1 = new Regex("(\\d+/\\w+/\\d+:\\d+)")

val p2 = new Regex("\\/twiki\\/bin\\/(\\w+)\\/")

val extractedData = logs.map { line =>
  val date = p1.findFirstIn(line).getOrElse("")
  val value = p2.findFirstIn(line).getOrElse("")
  (date, value)
}.filter{ case (date, value) => date != "" && value != ""}
```

Ce script scala traite de la même manière qu'en Map/Reduce les données du fichier logs. Ont définies les deux expressions régulières. Une fois ces expressions régulières établies, le code parcourt chaque ligne du fichier de logs. Pour chaque ligne, il cherche à extraire la date correspondant au format établi dans le regex et l'adresse IP définie par la seconde expression régulière. Si une correspondance est trouvée pour les deux, ces éléments sont retenus sinon la ligne est ignorée.

B- Analyse

Analyse par jour :

```
val day = extractedData.map{ case (date, value) =>
  val d = date.split(":")(0)
  (d, value)
}.map{ case (date, value) =>
  ((date, value), 1)
}.reduceByKey(_ + _)
```

Dans la variable day, les données extraite précédemment sont transformé pour ne garder que la date sans les heures. On réalise cela en divisant la data par le séparateur ' :' et en sélectionnant que le premier argument. Les valeurs sont regroupées par combinaison (date, adresse) associé à une valeur de 1 qui est en fait une sorte de compteur. Les clés sont ensuite agrégées avec l'addition des compteurs.

Résultat :

```
scala> day.collect().foreach(println)
((08/Mar/2004,view),166)
((08/Mar/2004,rename),3)
((12/Mar/2004,view),8)
((07/Mar/2004,rdiff),35)
((11/Mar/2004,edit),6)
((09/Mar/2004,oops),7)
((09/Mar/2004,edit),2)
((11/Mar/2004,view),54)
((09/Mar/2004,view),56)
((08/Mar/2004,oops),20)
((08/Mar/2004,rdiff),52)
((10/Mar/2004,view),39)
((07/Mar/2004,search),14)
((10/Mar/2004,oops),8)
((08/Mar/2004,search),22)
((07/Mar/2004,statistics),1)
((07/Mar/2004,attach),10)
((07/Mar/2004,view),55)
((11/Mar/2004,search),2)
((11/Mar/2004,rdiff),7)
((11/Mar/2004,oops),9)
((07/Mar/2004,edit),38)
((07/Mar/2004,oops),13)
((12/Mar/2004,oops),2)
((08/Mar/2004,attach),9)
((08/Mar/2004,edit),55)
```

Analyse par heure :

```
val hour = extractedData.map{ case (date, value) =>
  val h = date.split(":")(1)
  (h, value)
}.map{ case (date, value) =>
  ((date, value), 1)
}.reduceByKey(_ + _)
```

Dans la variable hour, de la même façon que pour la variable day, on ne va garder que l'heure. On réalise cela en divisant la data par le séparateur ' :' et en sélectionnant que le deuxième argument. Les valeurs sont regroupées par combinaison (heure, adresse) associé à une valeur de 1 qui est en fait une sorte de compteur. Les clés sont ensuite agrégées avec l'addition des compteurs.

Extrait du Résultat :

```
scala> hour.collect().foreach(println)
((07,rename),1)
((01,rdiff),2)
((02,rdiff),4)
((08,rdiff),4)
((06,view),17)
((04,view),16)
((07,view),16)
((02,oops),3)
((00,edit),2)
((10,search),1)
((19,view),10)
((05,edit),5)
((05,attach),1)
((15,search),2)
((12,oops),3)
((02,search),1)
((23,edit),3)
((20,oops),1)
((16,rdiff),6)
((22,view),13)
((13,attach),4)
((08,view),48)
((04,edit),6)
((00,view),9)
((13,edit),10)
```

III – PIG

A- Nettoyage des données

```
logs = LOAD '/user/cloudera/logss' USING TextLoader AS (line:chararray);

extractedData = FOREACH logs GENERATE
                REGEX_EXTRACT(line, '(\\d+/\\w+/\\d+:\\d+)', 1) AS date:chararray,
                REGEX_EXTRACT(line, '\\/twiki\\/bin\\/(\\w+)\\/', 1) AS value:chararray;

filteredData = FILTER extractedData BY date != '' and value != '';

finalData = FOREACH filteredData GENERATE
            FLATTEN(STRSPLIT(date, ':', 2)) AS (day:chararray, hour:chararray),
            value;
```

Toujours avec un traitement similaire à ceux vu précédemment, on va importer le fichier logs. On va ensuite dans une nouvelle variable extraire avec du regex la date et l'adresse IP. On vérifie si les deux valeurs ne sont pas nul. Dans une nouvelle variable on split la date pour obtenir le jour, l'heure et l'adresse.

B- Analyse

Analyse par jour par adresse :

```
groupedByDay = GROUP finalData BY (day, value);

countPerDay = FOREACH groupedByDay GENERATE
              FLATTEN(group) AS (day, value),
              COUNT(finalData) AS count;
```

Dans un premier temps on va regrouper le jour et l'adresse pour obtenir une sorte de clé unique qui va nous servir pour compter le nombre de fois qu'une date est associé à une adresse. Ce qui nous permettra d'obtenir le nombre de hit par jour par adresse.

Résultat :

```
(07/Mar/2004,edit,38)
(07/Mar/2004,oops,13)
(07/Mar/2004,view,55)
(07/Mar/2004,rdiff,35)
(07/Mar/2004,attach,10)
(07/Mar/2004,search,14)
(07/Mar/2004,statistics,1)
(08/Mar/2004,edit,55)
(08/Mar/2004,oops,20)
(08/Mar/2004,view,166)
(08/Mar/2004,rdiff,52)
(08/Mar/2004,attach,9)
(08/Mar/2004,rename,3)
(08/Mar/2004,search,22)
(09/Mar/2004,edit,2)
(09/Mar/2004,oops,7)
(09/Mar/2004,view,56)
(10/Mar/2004,oops,8)
(10/Mar/2004,view,39)
(11/Mar/2004,edit,6)
(11/Mar/2004,oops,9)
(11/Mar/2004,view,54)
(11/Mar/2004,rdiff,7)
(11/Mar/2004,search,2)
(12/Mar/2004,oops,2)
(12/Mar/2004,view,8)
```

Analyse par heure par sexe :

```
groupedByHour = GROUP finalData BY (hour, value);

countPerHour = FOREACH groupedByHour GENERATE
               FLATTEN(group) AS (hour, value),
               COUNT(finalData) AS count;
```

De la même façon, on va cette fois regrouper par heure et adresse et faire le comptage par heure et par adresse.

Extrait du Résultat :

```
(00,edit,2)
(00,oops,5)
(00,view,9)
(00,rdiff,2)
(00,attach,1)
(00,search,1)
(01,edit,6)
(01,oops,2)
(01,view,17)
(01,rdiff,2)
(01,search,2)
(02,edit,3)
(02,oops,3)
(02,view,10)
(02,rdiff,4)
(02,search,1)
(03,edit,4)
(03,oops,2)
(03,view,18)
(03,rdiff,5)
(03,search,1)
(04,edit,6)
(04,view,16)
(04,rdiff,5)
(05,edit,5)
(05,oops,1)
(05,view,30)
(05,rdiff,2)
(05,attach,1)
(05,search,2)
```

```
(06,edit,2)
(06,oops,1)
(06,view,17)
(06,rdiff,4)
(06,attach,2)
(07,edit,6)
(07,view,16)
(07,rdiff,2)
(07,rename,1)
(07,search,1)
(08,edit,3)
(08,view,48)
(08,rdiff,4)
(08,attach,1)
(08,search,1)
(09,edit,3)
(09,oops,5)
(09,view,9)
(09,rdiff,3)
(09,search,3)
(10,edit,2)
(10,oops,1)
(10,view,12)
(10,rdiff,3)
(10,rename,1)
(10,search,1)
(11,edit,6)
(11,oops,1)
(11,view,13)
(11,rdiff,2)
```

```
(11,rename,1)
(11,search,2)
(12,edit,2)
(12,oops,3)
(12,view,18)
(12,rdiff,4)
(12,search,3)
(13,edit,10)
(13,oops,4)
(13,view,38)
(13,rdiff,10)
(13,attach,4)
(13,search,3)
(14,edit,3)
(14,oops,2)
(14,view,15)
(14,rdiff,4)
(14,search,1)
(15,oops,2)
(15,view,3)
(15,rdiff,3)
(15,search,2)
(16,edit,4)
(16,oops,3)
(16,view,13)
(16,rdiff,6)
(16,attach,2)
(17,edit,4)
(17,oops,5)
```