Chapitre 4 :
Mise en œuvre de MapReduce sous Hadoop

4.1. Architecture MapReduce de Hadoop

Comme pour HDFS, la gestion des tâches de Hadoop se base sur deux serveurs (des daemons):

- Le JobTracker, qui va directement recevoir la tâche à exécuter (un .jar Java), ainsi que les données d'entrées (nom des fichiers stockés sur HDFS) et le répertoire où stocker les données de sortie (toujours sur HDFS). Il y a un seul JobTracker sur une seule machine du cluster Hadoop. Le JobTracker est en communication avec le NameNode de HDFS et sait donc où sont les données.
- Le TaskTracker, qui est en communication constante avec le JobTracker et va recevoir les opérations simples à effectuer (MAP/REDUCE) ainsi que les blocs de données correspondants (stockés sur HDFS). Il y a un TaskTracker sur chaque machine du cluster.

Comme le JobTracker est conscient de la position des données (grâce au NameNode), il peut facilement déterminer les meilleures machines auxquelles attribuer les sous-tâches (celles où les blocs de données correspondants sont stockés).

Pour effectuer un traitement Hadoop, on va donc stocker nos données d'entrée sur HDFS, créer un répertoire où Hadoop stockera les résultats sur HDFS, et compiler nos programmes MAP et REDUCE au sein d'un .jar Java.

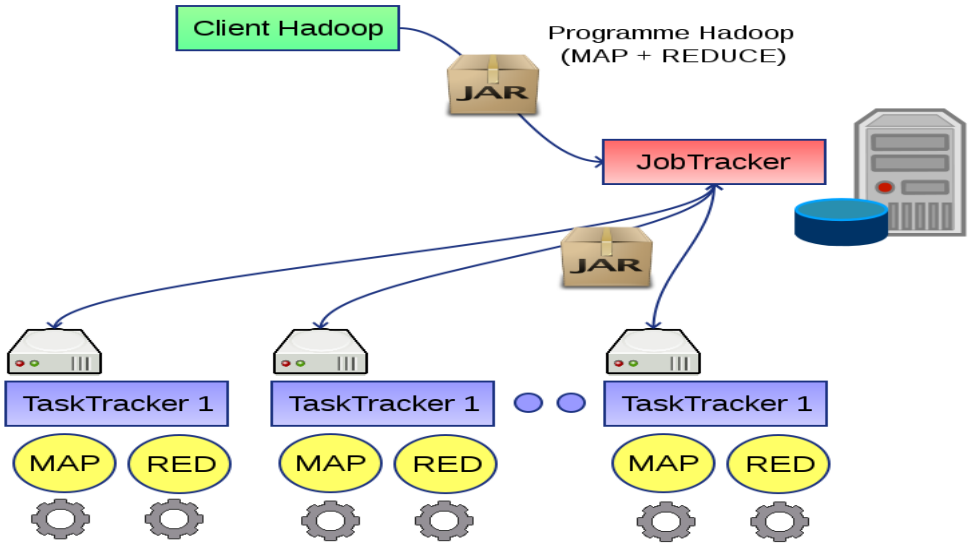

Fig. 4.1. Architecture logicielle MapReduce de Hadoop

On soumettra alors le nom des fichiers d'entrée, le nom du répertoire des résultats, et le .jar lui-même au JobTracker: il s'occupera du reste (et notamment de transmettre les programmes MAP et REDUCE aux serveurs TaskTracker des machines du cluster).

L'entrée de MapReduce doit être fait à partir de fichiers dans HDFS où Chaque bloc contient une liste de pairs clé-valeur.

4.1.1. Le JobTracker

Le déroulement de l'exécution d'une tâche Hadoop suit les étapes suivantes du point de vue du JobTracker :

1. Le client (un outil Hadoop console) va soumettre le travail à effectuer au JobTracker : une archive java .jar implémentant les opérations Map et Reduce. Il va également soumettre le nom des fichiers d'entrée et l'endroit où stocker les résultats.
2. Le JobTracker communique avec le NameNode HDFS pour savoir où se trouvent les blocs correspondant aux noms de fichiers donnés par le client.
3. Le JobTracker, à partir de ces informations, détermine quels sont les nœuds TaskTracker les plus appropriés, c'est à dire ceux qui contiennent les données sur lesquelles travailler sur la même machine, ou le plus proche possible (même rack/rack proche).
4. Pour chaque « morceau » des données d'entrée, le JobTracker envoie au TaskTracker sélectionné le travail à effectuer (MAP/REDUCE, code Java) et les blocs de données correspondants.
5. Le JobTracker communique avec les noeuds TaskTracker en train d'exécuter les tâches. Ils envoient régulièrement un **« heartbeat »**, un message signalant qu'ils travaillent toujours sur la sous-tâche reçue. Si aucun heartbeat n'est reçu dans une période donnée, le JobTracker considère la tâche comme ayant échouée et donne le même travail à effectuer à un autre TaskTracker.
6. Si par hasard une tâche échoue (erreur java, données incorrectes, etc.), le TaskTracker va signaler au JobTracker que la tâche n'a pas pu être exécutée.
7. Le JobTracker va alors décider de la conduite à adopter :
   - Demander au même TaskTracker de ré-essayer.
   - Redonner la sous-tâche à un autre TaskTracker.
   - Marquer les données concernées comme invalides, etc.
   - Il pourra même **blacklister** le TaskTracker concerné comme **non-fiable** dans certains cas.
8. Une fois que toutes les opérations envoyées aux TaskTracker (MAP + REDUCE) ont été effectuées et confirmées comme effectuées par tous les noeuds, le JobTracker marque la tâche comme « effectuée ». Des informations détaillées sont disponibles (statistiques, TaskTracker ayant posé problème, etc.).

Remarques:

Par ailleurs, on peut également obtenir à tout moment de la part du JobTracker des informations sur les tâches en train d'être effectuées: étape actuelle (MAP, SHUFFLE, REDUCE), pourcentage de complétion, etc.

4.1.2. Le TaskTracker

Le TaskTracker dispose d'un nombre de « slots » d'exécution. A chaque « slot » correspond une tâche exécutable (configurable). Ainsi, une machine ayant par exemple un processeur à 8 cœurs pourrait avoir 16 slots d'opérations configurées.

Lorsqu'il reçoit une nouvelle tâche à effectuer (MAP, REDUCE, SHUFFLE) depuis le JobTracker, le TaskTracker va démarrer une nouvelle instance de Java avec le fichier .jar fourni par le JobTracker, en appelant l'opération correspondante.

Une fois la tâche démarrée, il enverra régulièrement au JobTracker ses messages heartbeats. En dehors d'informer le JobTracker qu'il est toujours fonctionnels, ces messages indiquent également le nombre de slots disponibles sur le TaskTracker concerné.

Lorsqu'une sous-tâche est terminée, le TaskTracker envoie un message au JobTracker pour l'en informer, que la tâche se soit bien déroulée ou non (il indique évidemment le résultat au JobTracker)

4.1.3. Architecture Hadoop

De manière similaire au NameNode de HDFS, il n'y a qu'un seul JobTracker et s'il tombe en panne, le cluster tout entier ne peut plus effectuer de tâches. Des résolutions aux problèmes sont ajoutées dans la version 2 de Hadoop.

Généralement, on place le JobTracker et le NameNode HDFS sur la même machine (une machine plus puissante que les autres), sans y placer de TaskTracker/DataNode HDFS pour limiter la charge. Cette machine particulière au sein du cluster (qui contient les deux « gestionnaires», de tâches et de fichiers) est communément appelée le noeud maître («Master Node »). Les autres noeuds (contenant TaskTracker + DataNode) sont communément appelés noeuds esclaves (« slave node »).

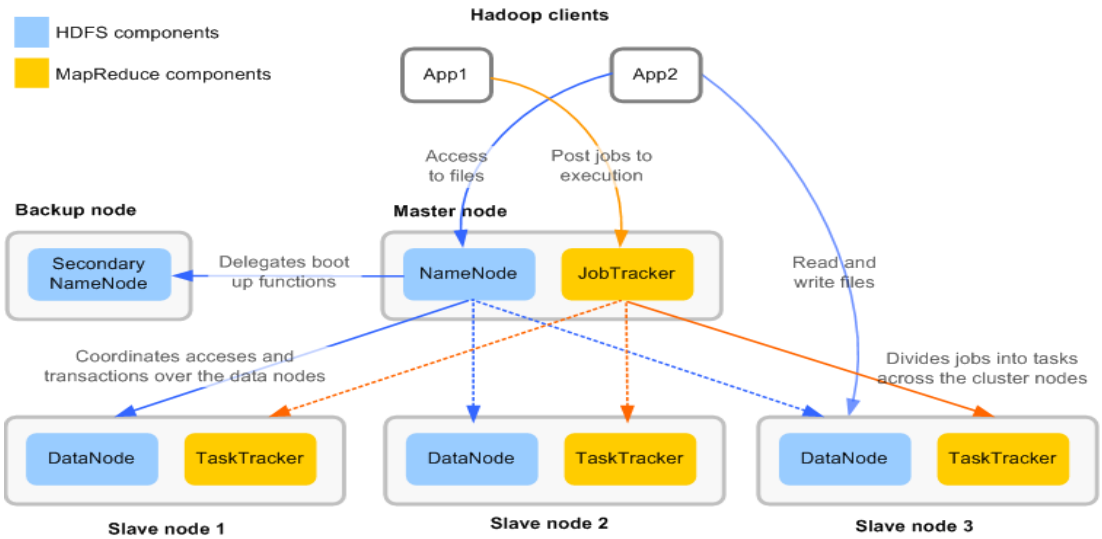

Fig. 4.2. Architecture Hadoop

4.2. Exemple illustratif

Les données: Les données utilisées pour cet exemple proviennent du National Climatic Data Center, les données sont stockées sous forme de lignes ASCII, où chaque ligne représente un enregistrement. Chaque enregistrement comporte un certain nombre d'informations météorologiques, certaines sont optionnelles ou de longueur variable, mais nous ne nous intéresserons qu'à la température qui est toujours présente et a une largeur de champ fixe.

```
0057
332130    # USAF weather station identifier
99999     # WBAN weather station identifier
19500101  # observation date
0300      # observation time
4
+51317    # latitude (degrees x 1000)
+028783   # longitude (degrees x 1000)
FM-12
+0171     # elevation (meters)
99999
V020
320       # wind direction (degrees)
1         # quality code
N
0072
1
00450     # sky ceiling height (meters)
1         # quality code
C
N
010000    # visibility distance (meters)
1         # quality code
N
9
-0128     # air temperature (degrees Celsius x 10)
1         # quality code
-0139     # dew point temperature (degrees Celsius x 10)
1         # quality code
10268     # atmospheric pressure (hectopascals x 10)
1         # quality code
```

Fig. 4.2. Format d'un enregistrement de la NCDC

Les fichiers de données sont organisés par date et par station d'observation. Il y'a un répertoire pour les données pour chaque année. Exemple pour 1990:

% **ls raw/1990 | head**

010010-99999-1990.gz

010014-99999-1990.gz

010015-99999-1990.gz

010016-99999-1990.gz

010017-99999-1990.gz

010030-99999-1990.gz

010040-99999-1990.gz

010080-99999-1990.gz

010100-99999-1990.gz

010150-99999-1990.gz

Pour cet exemple les données sont présentées à la fonction Map sous forme de paire clé-valeur (offset/ligne):

(0, 006701199099999**1950**051507004…9999999N9+**0000**1+99999999999…)

(106, 004301199099999**1950**051512004…9999999N9+**0022**1+99999999999…)

(212, 004301199099999**1950**051518004…9999999N9-**0011**1+99999999999…)

(318, 004301265099999**1949**032412004…0500001N9+**0111**1+99999999999…)

(424, 004301265099999**1949**032418004…0500001N9+**0078**1+99999999999…)

La fonction Map va seulement extraire la température pour chaque année.

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

…

Avant d'être envoyé au Reducer, la sortie de la fonction Map sont groupés et triés par clé/valeur. On obtient pour l'exemple précédent:

(1949, [111, 78])

(1950, [0, 22, −11])

Parmi ces listes les fonctions reduce renvoient le maximum.

(1949, 111)

(1950, 22)

4.3. Mise en œuvre sous Hadoop

La programmation d'un job MapReduce en Java nécessite de définir trois classes :

- Une sous-classe de **Mapper**. Elle contient une seule méthode, appelée map qui reçoit une paire clé-valeur en paramètre. Elle génère un nombre quelconque de paires.
- Une sous-classe de **Reducer**. Elle contient également une seule méthode, appelée reduce qui reçoit une liste de paires en paramètre. Elle génère une seule paire.
- Une classe générale qui crée un **Job** faisant référence aux deux précédentes classes.

Les deux premières sont des patrons (*templates*) paramétrées par les types des clés et des valeurs.

4.3.1. Les types de données MapReduce

La classe Mapper est paramétrée par 4 types. Ce ne sont pas les types standard de Java, mais des types spéciaux permettant de transmettre efficacement des données entre les différents ordinateurs du *cluster*.

Type	Description
Text	Chaîne UTF-8
BooleanWritable	Booléen
IntWritable	Entier 32 bits
LongWritable	Entier 64 bits
FloatWritable	Réel 32 bits
DoubleWritebale	Réel 64 bits

Les types Text, IntWritable. . . sont des implémentations d'une interface appelée Writable. Cette interface comprend :

- un constructeur. On peut mettre la valeur initiale en paramètre. IntWritable

val = new IntWritable(34);

- un modificateur : void set(*nouvelle valeur*);

val.set(35);

- un accesseur : *type* get()

int v = val.get();

L'interface Writable permet la *sérialisation*, c'est à dire l'écriture d'une structure de données sous forme d'octets et l'opération inverse, la *dé-sérialisation* qui permet de reconstruire une structure de données à partir d'octets.

La sérialisation est nécessaire pour échanger des données entre machines. On ne peut pas simplement échanger les octets internes car les machines du cluster ne sont pas obligatoirement toutes pareilles : nombre d'octets, ordre des octets. . .

Writable contient deux méthodes :

- void write(DataOutput out) : sérialisation
- void readFields(DataInput in) : dé-sérialisation

La classe Text permet de représenter n'importe quelle chaîne. Elle possède quelques méthodes à connaître :

- String toString() extrait la chaîne Java
- int getLength() retourne la longueur de la chaîne
- int charAt(int position) retourne le code UTF8 (appelé *point*) du caractère présent à cette position

La méthode reduce reçoit une collection de valeurs venant du *Mapper*. CleI et ValeursI sont les clés et valeurs intermédiaires. Il faut itérer sur chacune pour produire la valeur de sortie du réducteur.

Comme pour map, la classe est paramétrée par les types des clés et des valeurs à manipuler. Ce sont des Writable : Text, IntWritable. . .

Il est obligatoire que les types des clés TypCleI et valeurs d'entrée TypValI du réducteur soient exactement les mêmes que les types des clés et valeurs de sortie du *mapper*. Si vous mettez des types différents, ça passe à la compilation mais il y'aura erreur à l'exécution.

4.3.2. La classe Map

La fonction Map est structuré selon le format suivant :

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

Pour l'exemple de la température maximale, la fonction Map s'écrit en Java, comme suit :

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

4.3.3. La fonction Reduce

La fonction Map est structuré selon le format suivant :

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

Pour l'exemple de la température maximale, la fonction Reduce s'écrit en Java, comme suit :

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

4.3. La classe principale

Dans un Job MapReduce les classes Map et Reduce sont utilisé via la classe qui implémente le Job. Voici la classe principale qui crée et lance le job MapReduce :

```
public class Traitement extends Configured implements Tool
{
        public int run(String[] args) throws Exception
        {
                /* */
        }
        public static void main(String[] args) throws Exception
        {
                if (args.length != 2) System.exit(-1);
                Traitement traitement = new Traitement();
                System.exit( ToolRunner.run(traitement, args) );
        }
}
```

Pour l'exemple de la température maximale, la fonction « main » s'écrit en Java, comme suit :

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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 MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

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

     job.setMapperClass(MaxTemperatureMapper.class);
     job.setReducerClass(MaxTemperatureReducer.class);

     job.setOutputKeyClass(Text.class);
     job.setOutputValueClass(IntWritable.class);

     System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Un objet Job constitue la spécification du job et vous donne le contrôle sur la façon dont le job est exécuté. Lorsque nous exécutons ce job sur un cluster Hadoop, nous le paquetons dans un fichier JAR (que Hadoop distribuera au travers du cluster). Plutôt que de spécifier explicitement nom du fichier JAR, nous pouvons passer une classe dans la méthode setJarByClass () du Job, que Hadoop utilisera pour localiser le fichier JAR approprié en recherchant le fichier JAR contenant cette classe.

Après avoir construit un objet Job, nous spécifions les chemins d'entrée et de sortie. Un chemin d'entrée est spécifié en appelant la méthode statique addInputPath () sur FileInputFormat, et il peut être un seul fichier, un répertoire (dans ce cas, l'entrée forme tous les fichiers dans ce répertoire), ou un modèle de fichier. Comme son nom l'indique, addInputPath () peut être appelé plus d'une fois pour utiliser des entrées provenant de plusieurs chemins.

Le chemin de sortie (dont il n'y en a qu'un) est spécifié par la static setOutputPath () méthode sur FileOutputFormat. Il spécifie un répertoire où les fichiers de sortie du Reducer sont écrits. Le répertoire ne doit pas exister avant d'exécuter le travail, car Hadoop ne va pas exécuter le travail. Cette précaution vise à prévenir la perte de données.

Ensuite, nous spécifions les types du Mapper et du Reducer via setMapperClass () et Méthodes setReducerClass ().

Les méthodes setOutputKeyClass () et setOutputValueClass () contrôlent les types de sortie pour la fonction Reduce, et doit correspondre à ce que la classe Reduce produit. La sortie de la fonction Map sont par défaut ceux de la fonction Reduce, ainsi si le Mapper produit le m^me type de sorite que le Reducer, ils n'ont pas besoin d'être définis (comme c'est le cas ici). Cependant, s'ils sont différents, les types de sortie du Map doivent être définis à l'aide de setMapOutputKeyClass () et Méthodes setMapOutputValueClass ().

Les types d'entrée sont contrôlés via le format d'entrée, que nous n'avons pas explicitement défini parce que nous utilisons le TextInputFormat par défaut.

Après avoir défini les classes qui définissent les fonctions Map et Reduce, nous sommes prêts à exécuter le job. La méthode waitForCompletion () sur Job soumet le travail et attend qu'il se termine.

Le seul argument de la méthode est un drapeau indiquant si la sortie verbeuse est généré. Lorsque la valeur est true, le travail écrit des informations sur sa progression dans la console.

La valeur de retour de la méthode waitForCompletion () est un booléen indiquant le succès (true) ou failure (false), que nous traduisons par le code de sortie 0 ou 1 du programme.

Lancement du job:

```
$ export HADOOP_CLASSPATH=hadoop-examples.jar
$ hadoop MaxTemperature input/ncdc/sample.txt output
```