المدرسة الوطنية للعلوم التطبيقية
ⵜⵉⵎⵙⵙⵓⵜ ⵜⵓⵏⵙⵉⵢⵜ ⵜⵎⵙⵙⵓⵔⵉⵜ ⵜⵉⵡⵉⵔⵉⵡⵉⵏ
Ecole Nationale des Sciences Appliquées de Fès

جامعة سيدي محمد بن عبد الله
ⵜⵓⵏⵉⵡⵉⵙⵜ ⵓⵙⵍⵙ ⵉⵙⵎⵎⵓⵏ ⵙⵉ ⵎⵓⵃⴰⵎⵎⴻⴷ ⴱⵏ ⵄⴱⴷⵍⵍⴰ
Université Sidi Mohamed Ben Abdellah

Atelier: N°4

Apache Hadoop: MapReduce Applications

Pr. MOUNTASSER IMADEDDINE

Objectifs

L'augmentation et l'explosion du volume de données générées aujourd'hui posent un défi majeur à la gestion et au traitement de ces données. L'évolutivité, la fiabilité et la rentabilité du stockage et du traitement de ces données sont devenues des priorités pour les entreprises et les organisations depuis une décennie.

MapReduce est un **Framework** de **traitement de données** distribué utilisé dans Hadoop qui facilite le développement d'applications logicielles traitant d'importants volumes de données. MapReduce permet de répartir la charge de travail (calculs) entre différents nœuds (à l'instar du matériel standard). Le temps de traitement est ainsi réduit, car les données à traiter sont désormais divisées en petits blocs et traitées individuellement. MapReduce divise généralement les tâches en deux phases principales, phase de mappage et phase de réduction. La première consiste à séparer les données d'entrée en paires clé-valeur et les traiter ensuite indépendamment. La deuxième permet d'agréger les résultats mappés en fonction des clés en exécutant des opérations telles que le comptage, la sommation ou le filtrage.

En général, les nœuds de calcul et de stockage sont identiques. Cela signifie que MapReduce et HDFS s'exécutent sur le même ensemble de nœuds. Cette configuration permet au Framework de planifier efficacement les tâches sur les nœuds contenant déjà des données, ce qui génère une bande passante agrégée et très élevée sur l'ensemble du cluster. Au minimum, les applications spécifient les emplacements d'entrée/sortie et fournissent des fonctions de mappage et de réduction via des implémentations d'interfaces et/ou de classes abstraites appropriées. Ces paramètres, ainsi que d'autres paramètres du Job, constituent la configuration du Job. Le client soumet ensuite le Job (fichier JAR/exécutable, etc.) et la configuration au gestionnaire de ressources, qui se charge ensuite de distribuer le programme/la configuration aux travailleurs, de planifier les tâches et de les surveiller, et de fournir des informations d'état et de diagnostic au client du Job.

Le paradigme MapReduce est largement utilisé pour le traitement du Big Data, permettant une exécution parallèle sur plusieurs machines d'un cluster Hadoop. Ainsi, dans cet atelier, nous allons adopter le même paradigme par le développement d'un ensemble d'applications capables de traiter parallèlement des données à grande échelle.

I. Développement de l'Application WordCount

L'objectif du job WordCount est de compter le nombre d'occurrences de chaque mot dans un ensemble de documents. C'est le programme "Hello World" du paradigme MapReduce (Hadoop). En effet, c'est un exemple simple mais représentatif qui illustre de manière significative les fonctionnalités de base de MapReduce. Il est facile à implémenter et utilisé comme point de départ pour des jobs plus complexes : tri, join, group by, etc. Ce job traite un fichier texte donné et compte les occurrences de chaque mot.

Tout d'abord, nous allons présenter les étapes pour écrire du code MapReduce pour le comptage de mots. Généralement, le modèle MapReduce se base sur deux phases principales :

► **Phase Map**
→ Le Mapper lit les données ligne par ligne.
→ Chaque ligne est découpée en mots (i.e. tokenisation).
→ Pour chaque mot, la tâche émet une paire clé-valeur (mot, 1).

► **Phase Reduce**
→ Le Reducer reçoit chaque mot avec une liste de 1.
→ Il fait la somme des valeurs.

Il faut noter la présence de certaines phases intermédiaires qui participent au la bon exécution du Job (ex. Shffling et Sort).

► **Shuffle & Sort (interne à Hadoop)**
→ Hadoop regroupe toutes les paires par clé (mot).
→ Toutes les paires (mot, 1) sont regroupées sous une même clé.

I.1. Préparation de l'Entrée & Création du Projet

Passons maintenant à l'implémentation. Nous allons implémenter nos propres codes Mapper et Reducer. Nous allons d'abord importer notre jeu de données dans le système de fichiers distribués HDFS. Le jeu de données peut être un simple fichier texte contenant quelques mots ou phrases.

Veuillez suivre ces étapes pour importer l'ensemble de données :

- Créez un répertoire dans HDFS en utilisant la commande suivante :

```
hdfs dfs -mkdir /user/input_wordCount
```

- Copiez le fichier texte depuis le répertoire local vers le répertoire hdfs créé avec :

```
hdfs dfs -put C:/input.txt /user/input_wordCount/input
```

Nous allons maintenant préparer le code source pour le comptage de mots. Pour cela, il est fortement recommandé d'utiliser un IDE (i.e. IntelliJ) permettant de créer un fichier JAR exécutable pour notre projet.
A cette fin, nous allons construire un projet Java basé sur Maven pour écrire notre application WordCount. Commencez par créer un projet sur IntelliJ nommé WordCountProject. N'oubliez pas de préciser le JDK 11 compatible avec la version Hadoop utilisée.

Ensuite, procédez à modifier l'environnement du projet afin qu'il corresponde à notre application Hadoop cible. Commençons par configurer les dépendances de notre projet en ajustant le fichier pom.xml avec les dépendances requises (Similairement à ce que nous avons réalisé dans l'atelier 3.0). Nous devons aussi inclure les dépendances Hadoop MapReduce dans votre projet.

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>3.3.6</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-jobclient</artifactId>
    <version>3.3.6</version>
</dependency>
```

Il faut référencer les bibliothèques Hadoop afin qu'elles soient disponibles pour la programmation. La version utilisée doit correspondre à la version de Hadoop installée.

Maintenant, nous allons créer une classe qui nous permettra de se connecter à notre Hadoop local en cours d'exécution (utilisez la classe HDFSConnexion déjà créée dans l'atelier 3.0). Puis, nous allons créer trois classes Java, à savoir, WordCountDriver (ayant la fonction principale), WordCountMapper, WordCountReducer.

I.2. Création de la Class Mapper

La classe **WordCoundMapper** est la classe Mapper qui lit une ligne de texte, la découpe en mots, puis émet une paire (mot, 1) pour chaque mot trouvé. C'est la première étape du job MapReduce. La classe hérite de Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>. Dans notre cas, KEYIN représente la clé d'entrée, typiquement l'offset dans le fichier, VALUEIN représente Text, i.e. une ligne du fichier, KEYOUT de type Text représentant un mot, et VALUEOUT de type IntWritable représentant la valeur associée (ici 1) (**Figure. 1**).

Nous allons essayer de créer une constante **one** qui sera réutilisée pour chaque mot (cela évite d'instancier plusieurs objets) et une variable **word** qui contiendra chaque mot extrait. De plus, notre classe étend la classe Mapper déjà prédéfinie et fournie par Hadoop. Ainsi, nous nous concentrons uniquement sur la fonction map qui nous allons générer (Clic droit → Generate → Override Methodes → map).

```
package mr;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
import java.util.StringTokenizer;

public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //Version I
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
        //Version II
        /*String line = value.toString();
        String[] words=line.split(" ");
        for(String word: words){
            if (word.length() > 1){
                Text outputKey = new Text(word.toLowerCase().trim());
                IntWritable outputValue = new IntWritable(1);
                context.write(outputKey, outputValue);
             }    }*/
    }
}
```

Figure 1. Classe WordCountMapper

L'implémentation du mappeur, via la méthode map, traite une ligne à la fois, conformément au TextInputFormat spécifié. Elle divise ensuite la ligne en jetons séparés par des espaces, via

le StringTokenizer (découpe la ligne en mots individuels selon les espaces), et génère une paire clé-valeur < <word>, 1>.

Remarque :

Vous pouvez ajouter des améliorations (**Figure. 2**) à votre code en ajoutant des étapes de :

- **Normalisation** : Convertir en minuscules les mots via word.set(itr.nextToken().toLowerCase()).
- **Nettoyage** : Retirer et éliminer la ponctuation (avec regex).
- **Filtrage** : Ignorer les mots vides (stop words).

```
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //Version III
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            // Nettoyage : minuscules et suppression des caractères non alphabétiques (Ponctuation)
            String cleanedWord = itr.nextToken().toLowerCase().replaceAll("[^a-z]", "");
            // Filtrage : ignorer les mots vides et les stop words
            if (!cleanedWord.isEmpty() && !STOP_WORDS.contains(cleanedWord)) {
                word.set(cleanedWord);
                context.write(word, one);
            }
        }
    }
}
```

Figure 2. WordCountMapper Amélioré

I.3. Création de la Class Reducer

La classe **WordCountReducer** est le Reducer de notre job MapReduce. Elle reçoit, pour chaque mot unique, la liste des valeurs associées (des 1) émises par le Mapper, et calcule leur somme totale, c'est-à-dire le nombre d'occurrences de ce mot dans l'ensemble des données. La classe hérite de Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>. Dans notre cas, KEYIN représente la clé d'entrée de type Text, typiquement le mot, VALUEIN représente les valeurs d'entrée de type IntWritable (i.e. les 1 associés), KEYOUT de type Text représentant un mot, et VALUEOUT de type IntWritable représentant le total d'occurrences (**Figure. 3**).

Nous allons essayer de créer un objet IntWritable réutilisable pour contenir le total des occurrences d'un mot **result**. De même notre classe étend la classe Reducer déjà prédéfinie et fournie par Hadoop. Ainsi, nous nous concentrons uniquement sur la fonction reduce qui nous allons générer (Clic droit → Generate → Override Methodes → reduce).

En fait, l'implémentation du reducer appelle automatiquement la méthode reduce () pour chaque mot unique (key) avec la liste de toutes les valeurs 1 associées à ce mot.

```java
package mr;

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


import java.io.IOException;
import java.util.Iterator;

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

    private IntWritable result = new IntWritable();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Figure 3. Classe WordCountReducer

I.4. Création de la Class Driver

Globalement, la classe Driver joue le rôle de chef d'orchestre. Elle spécifie les différentes facettes du Job. Elles responsable de configurer tous les composants nécessaires pour exécuter un traitement MapReduce (Mapper et Reducer à utiliser, les types de données en entrée/sortie, et les chemins d'entrée et de sortie sur HDFS, etc.). Ces chemins sont transmis via la ligne de commande.

En effet, nous pouvons dire qu'elle prépare l'exécution du Job sur le cluster en informant Hadoop de quelle classe contient la méthode main() (avec setJarByClass()), où sont situés les fichiers d'entrée (avec FileInputFormat.addInputPath()) et où écrire les résultats (avec FileOutputFormat.setOutputPath()). Finalement, elle lance le job en appelant la méthode job.waitForCompletion pour soumettre le job et suivre sa progression.

Ainsi, nous pouvons considérer la classe Driver comme un chef de projet ; elle ne fait pas elle-même le traitement, mais elle organise et coordonne les étapes, les rôles (Mapper/Reducer), les ressources (fichiers), et surveille l'exécution jusqu'à la fin.

En général, les implémentations de mappeurs sont transmises au job via la méthode Job.setMapperClass(Mapper.Class). Le Framework appelle ensuite map(WritableComparable, Writable, Context) pour chaque paire clé/valeur de l'InputSplit pour cette tâche. En effet, MapReduce lit les fichiers en parallèle et automatiquement via FileInputFormat. Toutes les valeurs intermédiaires associées à une clé de sortie donnée sont ensuite regroupées par le Framework et transmises au(x) Reducer(s) pour déterminer la sortie finale. Les utilisateurs peuvent contrôler le regroupement en spécifiant un Comparator via Job.setGroupingComparatorClass(Class).

Les sorties du mappeur sont triées puis partitionnées par réducteur. Le nombre total de partitions est identique au nombre de tâches de réduction pour le job. Les utilisateurs peuvent contrôler quelles clés (et donc quels enregistrements) sont dirigés vers quel réducteur en implémentant un partitionneur personnalisé. Les utilisateurs peuvent éventuellement spécifier un combinateur, via Job.setCombinerClass(Class), pour effectuer une agrégation locale des sorties intermédiaires, ce qui permet de réduire la quantité de données transférées du mappeur vers le réducteur.

Dans notre classe Driver, vous utiliserez la classe HDFSConnexion pour récupérer l'objet conf. Ensuite, vous créez une instance de Job, qui représente une exécution MapReduce et spécifiez la classe contenant la méthode main(), nécessaire pour le déploiement du JAR. Après cela, vous devez définir les classes Mapper et Reducer que le job doit utiliser avec la possibilité d'effectuer une agrégation locale avant le shuffle via un Combiner (même Reducer). Finalement, vous devez définir les types de la sortie du Mapper/Reducer, i.e. Text pour les mots, IntWritable pour les comptes, ainsi que préciser les fichiers ou dossiers d'entrée et sortie sur HDFS. Le traitement de ce job est lancé via job.waitForCompletion(true) lance le traitement MapReduce (la valeur true permet d'afficher les logs).

Dans notre fonction main(), la configuration Hadoop (conf) est initialisée pour lancer le job. Pour cela, nous allons découvrir deux alternatives. La première consiste à exécuter notre code en mode local (**Figure. 4**). Cela exécutera notre job localement, dans le même processus que notre IDE, donc il trouvera toutes les classes de notre projet sans créer de jar. C'est une implémentation plus rapide destinée au test/débogage sans créer un jar ou démarrer les services start et yarn (start-dfs.cmd & start-yarn.cmd). Mais, elle est limitée à de petits tests, sans bénéficier des performances d'un vrai cluster YARN.

```java
package mr;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

    private HDFSConnexion connexion;

    public WordCountDriver() throws Exception {
        connexion = new HDFSConnexion();
    }

    public HDFSConnexion getConnexion() {
        return connexion;
    }

    public static void main(String[] args) throws Exception {
        String inputPath = "/user/input_wordCount";
        String outputPath = "/user/output_wordCount/output";

        WordCountDriver WCD = new WordCountDriver();
        //  Création de la configuration Hadoop
        Configuration conf = WCD.getConnexion().getFS().getConf();
        conf.set("mapreduce.framework.name", "local");
        conf.set("mapreduce.cluster.local.dir", "C:/hadoop_tmp/local");
        //System.out.println("Connected to HDFS! : (" + WCD.getConnexion().getFS().getConf().toString()+")");
        // Création d'une instance Job
        Job job = Job.getInstance(conf, "Word Count");
        // Indiquer la classe principale (celle contenant le main)
        job.setJarByClass(WordCountDriver.class);
        // Spécifier les classes Mapper et Reducer
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        // (Optionnel) Pour plus de performance, spécifier aussi le Combiner (ex. identique au Reducer)
        job.setCombinerClass(WordCountReducer.class);
        // Définir les types de sortie (clé/valeur)
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // Définir les chemins d'entrée et de sortie HDFS
        FileInputFormat.addInputPath(job, new Path(inputPath));
        FileOutputFormat.setOutputPath(job, new Path(outputPath));
        // Lancer le job et attendre sa fin
        System.exit(job.waitForCompletion(true) ? 0 : 1);

    }
}
```

Figure 4. Classe Driver en Mode Local

La deuxième alternative consiste à générer automatiquement le jar à chaque exécution (**Figure. 5**). Cela passe par configurer un plugin Maven dans notre IDE pour générer un .jar contenant toutes les classes nécessaires à l'exécution de notre job (via hadoop -jar en CLI). Dans ce cas, le job MapReduce est exécuté en mode YARN (cluster ou pseudo-distribué) et c'est l'alternative la plus recommandée et la plus fiable de s'assurer que notre code Java est visible par le cluster YARN.

```java
package mr;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

    private HDFSConnexion connexion;

    public WordCountDriver() throws Exception {
        connexion = new HDFSConnexion();
    }

    public HDFSConnexion getConnexion() {
        return connexion;
    }

    public static void main(String[] args) throws Exception {

        // Vérification des arguments : 2 requis (input + output)
        if (args.length != 2) {
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(-1);
        }
       WordCountDriver WCD = new WordCountDriver();
       //  Création de la configuration Hadoop
       Configuration conf = WCD.getConnexion().getFS().getConf();
       conf.set("mapreduce.framework.name", "local");
       conf.set("mapreduce.cluster.local.dir", "C:/hadoop_tmp/local");
       //System.out.println("Connected to HDFS! : (" + WCD.getConnexion().getFS().getConf().toString()+")");
       // Création d'une instance Job
       Job job = Job.getInstance(conf, "Word Count");
       // Indiquer la classe principale (celle contenant le main)
       job.setJarByClass(WordCountDriver.class);
       // Spécifier les classes Mapper et Reducer
       job.setMapperClass(WordCountMapper.class);
       job.setReducerClass(WordCountReducer.class);
       // (Optionnel) Pour plus de performance, spécifier aussi le Combiner (ex. identique au Reducer)
       job.setCombinerClass(WordCountReducer.class);
       // Définir les types de sortie (clé/valeur)
       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(IntWritable.class);
       // Définir les chemins d'entrée et de sortie HDFS
       FileInputFormat.addInputPath(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));
       // Lancer le job et attendre sa fin
       System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 5. Classe Driver en Mode YARN

Normalement, pour créer un fichier JAR automatiquement avec Maven (à partir de votre fichier pom.xml), vous devez ajouter un plugin Maven (maven-jar-plugin/ maven-shade-plugin) (Voir Atelier 3.0). Pour réaliser le packaging. Il faut noter que nous devons ajouter un plugin de compilation (compiler) Maven et inclure également une classe principale pour le fichier JAR exécutable (**Figure. 6**).

```
<build>
   <plugins>
   <!-- Compiler plugin -->
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>3.10.1</version>
         <configuration>
            <source>11</source>
            <target>11</target>
         </configuration>
       </plugin>
 <!-- Shade plugin: to create an executable fat JAR (Uber Jar) -->
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-shade-plugin</artifactId>
         <version>3.4.1</version>
         <executions>
            <execution>
               <phase>package</phase>
               <goals><goal>shade</goal></goals>
              <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
                <transformers>
                    <!-- Merge META-INF/services properly -->
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                     <mainClass>mr.WordCountDriver</mainClass>
                     </transformer>
                </transformers>
              </configuration>
            </execution>
          </executions>
      </plugin>
   </plugins>
 </build>
```

Figure 6. Configuration à ajouter au Fichier pom.xml

Pour résumé, pendant le développement, vous pouvez utiliser (mapreduce.framework.name = local) dans notre code pour tester dans l'IDE. Quand vous passez à l'exécution réelle (i.e. production) sur HDFS/YARN, il faut générer un .jar et l'exécute via hadoop jar.

Maintenant, exécutez l'application en spécifiant maintenant les arguments via la ligne de commande comme suit :

```
hadoop -jar path_to_target/WordCount_Project-1.0.0.jar
hdfs://localhost:9000/user/folder_1/logs /user/output_wordCount/output1
```

II. Développement de l'Application UserLogsCounter

L'objectif du job UserLogsCounter est de compter le nombre de lignes associées à chaque utilisateur à partir de logs. Cette application nous permet de mesurer l'activité ou l'engagement des utilisateurs dans un système à grande échelle (web, mobile, réseau, etc.). Ce type d'analyse est souvent une étape de base mais cruciale dans le traitement de logs pour en tirer des indicateurs analytiques.

Généralement, mesurer l'activité utilisateur consiste à calculer le nombre d'actions effectuées par utilisateur (ex. vues de pages, clics, connexions) afin de, par exemple, détecter des comportements inhabituels. Par exemple, si on constate un nombre anormalement élevé de lignes par un utilisateur, cela peut indiquer un bot ou une attaque. Ce genre d'application assurera aussi le suivi de la répartition de la charge ou de l'utilisation, en vue d'identifier les utilisateurs les plus actifs pour optimiser les ressources.

Exemples :

- **Sites e-commerce (ex: Amazon, Cdiscount) :**
  - Compter les interactions des utilisateurs avec les produits.
  - Ex. Combien de pages vues par utilisateur ? Combien d'ajouts au panier ?
- **Réseaux sociaux (ex: Facebook, Twitter)**
  - Suivre le nombre d'actions (like, share, commentaire) par utilisateur.
  - Identifier les utilisateurs les plus actifs ou influents.
- **Plateformes de streaming (ex: Netflix, Spotify)**
  - Compter le nombre de contenus consultés ou lus.
  - Ex. Combien de vidéos ou morceaux écoutés par utilisateur par jour ?
- **Systèmes d'authentification (ex: SSO d'entreprise)**
  - Analyser les logs de connexion pour détecter les pics d'accès ou tentatives suspectes.
  - Ex. 50 connexions par minute d'un seul utilisateur => possible brute-force.
- **Applications mobiles (via Firebase ou logs internes)**
  - Compter les événements enregistrés dans les logs (clics, vues d'écrans).
  - Permet de connaître les usages réels de certaines fonctionnalités.

II.1. Préparation de l'Entrée & Création du Projet

Passons maintenant à l'implémentation. Nous allons implémenter nos propres codes Mapper et Reducer. Nous allons d'abord importer notre jeu de données dans le système de fichiers distribués HDFS. Le jeu de données contient plusieurs logs sous la forme suivante :

```
2025-05-08 10:12:01 user1 login success
2025-05-08 10:12:05 user2 login failed
2025-05-08 10:13:10 user1 view page home
2025-05-08 10:14:20 user3 logout
2025-05-08 10:15:00 user2 view page contact
```

Veuillez suivre ces étapes pour importer l'ensemble de données :

- Créez un répertoire dans HDFS en utilisant la commande suivante :

```
hdfs dfs -mkdir /user/users_logs
```

- Copiez le fichier texte depuis le répertoire local vers le répertoire hdfs créé avec :

```
hdfs dfs -put C:/logs_1000.txt /user/users_logs/input
```

De la même manière que l'application WordCount, nous allons construire un projet Java basé sur Maven pour écrire notre application **UserLogsCounter**, configurer les dépendances en ajustant le fichier pom.xml avec les dépendances requises, et procéder à créer nos trois classes Java, UserLogsCounterDriver (ayant la fonction principale), UserLogsCounterMapper, UserLogsCounterReducer.

II.2. Création de la Class Mapper

La classe **UserLogsCounterMapper** est la classe Mapper, responsable d'extraire, pour chaque ligne de log, l'identifiant de l'utilisateur concerné, puis d'émettre une paire clé-valeur (utilisateur, 1) indiquant que cet utilisateur a généré une ligne (**Figure. 7**).

```java
package mr;

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


public class UserLogsCounterMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text username = new Text();

    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        // Chaque ligne de log
        String line = value.toString();
        String[] tokens = line.split(" ");
        if (tokens.length >= 3) {
            String user = tokens[2]; // user1, user2, etc.
            username.set(user);
            context.write(username, one);
        }
    }
}
```

Figure 7. Classe UserLogsCounterMapper

II.3. Création de la Class Reducer

La classe **UserLogsCounterReducer** a pour objectif de regrouper toutes les paires (clé = utilisateur, valeur = 1) générées par le Mapper, puis de calculer la somme de ces 1 pour chaque utilisateur (**Figure. 8**).

```java
package mr;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

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

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Figure 8. Classe UserLogsCounterReducer

II.4. Création de la Class Driver

Similairement à la première application, la classe UserLogsCounterDriver configure et déclenche l'exécution de notre job MapReduce. Elle prépare la configuration Hadoop, définit les classes Mapper, Reducer (et éventuellement Combiner), spécifie les types de clés/valeurs et définit les chemins d'entrée/sortie (HDFS ou local selon le contexte) (**Figure. 9**).

```java
package mr;

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

public class UserLogsCounterDriver {

    private HDFSConnexion connexion;

    public UserLogsCounterDriver() throws Exception {
        connexion = new HDFSConnexion();
    }

    public HDFSConnexion getConnexion() { return connexion; }

    public static void main(String[] args) throws Exception {

        // Vérification des arguments : 2 requis (input + output)
        if (args.length != 2) {
            System.err.println("Usage: UserLogsCounter <input path> <output path>");
            System.exit(-1);
        }
        UserLogsCounterDriver ULCD = new UserLogsCounterDriver();
        //  Création de la configuration Hadoop
        Configuration conf = ULCD.getConnexion().getFS().getConf();
        Job job = Job.getInstance(conf, "Users Logs Count");
        // Indiquer la classe principale (celle contenant le main)
        job.setJarByClass(UserLogsCounterDriver.class);
        // Spécifier les classes Mapper et Reducer
        job.setMapperClass(UserLogsCounterMapper.class);
        job.setReducerClass(UserLogsCounterReducer.class);
        // (Optionnel) Pour plus de performance, spécifier aussi le Combiner (ex. identique au Reducer)
        job.setCombinerClass(UserLogsCounterReducer.class);
        // Définir les types de sortie (clé/valeur)
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // Définir les chemins d'entrée et de sortie HDFS
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        // Lancer le job et attendre sa fin
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 9. Classe UserLogsCounterDriver

III. Développement de l'Application AVGTempDay

Maintenant, nous allons implémenter un cas d'usage classique de MapReduce, i.e. agréger des données pour produire des statistiques. Ainsi, nous allons implémenter un job qui calcule la température moyenne par jour (ou par station si on adapte la clé).

III.1. Préparation de l'Entrée & Création du Projet

Important d'abord le jeu de données que notre job va traiter. Le jeu de données d'entrée représente des données météorologiques, sous la forme (Date, Station, Température), souvent collectées automatiquement par des capteurs situés dans des stations météorologiques.

```
2025-05-06,StationA,18.5
2025-05-06,StationA,19.2
2025-05-06,StationB,17.0
2025-05-07,StationA,20.1
```

En effet, nous allons importer un fichier contenant 180000 lignes de données météorologiques (180 jours × 1000 enregistrements par jour) dans le système de fichiers distribués HDFS. Pour cela, suivez ces étapes pour importer l'ensemble de données :

- Créez un répertoire dans HDFS en utilisant la commande suivante :

```
hdfs dfs -mkdir /user/stations_temp
```

- Copiez le fichier texte depuis le répertoire local vers le répertoire hdfs créé avec :

```
hdfs dfs -put C:/meteo_dataset.txt /user/stations_temp/input
```

Ensuite, de la même manière que l'application WordCount, nous allons construire un projet Java basé sur Maven pour écrire notre application AVGTempDay, configurer les dépendances en ajustant le fichier pom.xml avec les dépendances requises, et procéder à créer nos trois classes Java, AVGTempDayDriver (ayant la fonction principale), AVGTempDayMapper, AVGTempDayReducer.

III.2. Création de la Class Mapper

La classe **AVGTempDayMapper** est la classe Mapper, responsable d'extraire, pour chaque ligne, la date, la station en question et la température correspondante, puis de spécifier la clé que nous voulons utiliser pour calculer la moyenne (**Figure. 10**).

III.3. Création de la Class Reducer

La classe **AVGTempDayReducer** a pour objectif de regrouper toutes les paires générées par le Mapper, puis de calculer la moyenne des températures pour chaque clé (**Figure. 11**). Vous pouvez calculer aussi le min/max. Il suffit de modifier le Reducer.

III.4. Création de la Class Driver

La classe **AVGTempDayDriver** configure et déclenche l'exécution de notre job. Elle prépare la configuration Hadoop, définit les classes Mapper, Reducer (et éventuellement Combiner), spécifie les types de clés/valeurs de sortie (dans ce cas, FloatWritable) et définit les chemins d'entrée/sortie (HDFS ou local selon le contexte) (**Figure. 12**).

```java
package mr;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

public class AVGTempDayMapper extends Mapper<Object, Text, Text, FloatWritable> {

    private Text outputKey = new Text();
    private FloatWritable outputValue = new FloatWritable();

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

        String line = value.toString();
        String[] fields = line.split(",");

        if (fields.length == 3) {
            String date = fields[0].trim();
            String station = fields[1].trim();
            float temp = Float.parseFloat(fields[2].trim());
            // Maintenant, faisons notre choix de la clé de groupement : date seule ou station+date
            // Par jour uniquement :
            outputKey.set(date);
            // Pour groupement par station :
            // outputKey.set(station);
            outputValue.set(temp);
            context.write(outputKey, outputValue);
        }
    }
}
```

Figure 10. Classe AVGTempDayMapper

```java
package mr;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class AVGTempDayReducer extends Reducer<Text, FloatWritable, Text, FloatWritable> {

    private FloatWritable result = new FloatWritable();

    @Override
    protected void reduce(Text key, Iterable<FloatWritable> values, Context context) throws IOException, InterruptedException {
        float sum = 0;
        int count = 0;

        for (FloatWritable val : values) {
            sum += val.get();
            count++;
        }
        if (count > 0) {
            float average = sum / count;
            result.set(average);
            context.write(key, result);
        }
    }
}
```

Figure 11. Classe AVGTempReducer

```java
package mr;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
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 AVGTempDayDriver {

    private HDFSConnexion connexion;

    public AVGTempDayDriver() throws Exception {
        connexion = new HDFSConnexion();
    }

    public HDFSConnexion getConnexion() { return connexion; }

    public static void main(String[] args) throws Exception {
        // Vérification des arguments : 2 requis (input + output)
        if (args.length != 2) {
            System.err.println("Usage: AverageTemperature <input path> <output path>");
            System.exit(-1);
        }
        AVGTempDayDriver ATDD = new AVGTempDayDriver();
        Configuration conf = ATDD.getConnexion().getFS().getConf();
        Job job = Job.getInstance(conf, "AVGTempDay");
        job.setJarByClass(AVGTempDayDriver.class);
        job.setMapperClass(AVGTempDayMapper.class);
        job.setReducerClass(AVGTempDayReducer.class);
        job.setCombinerClass(AVGTempDayReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 12. Classe AVGTempDayDriver