Projet Hagidoop
Une implantation du modèle Map-Reduce en Java

Daniel Hagimont

ENSEEIHT Département SN, 2ième année, 2023-2024

Le but de ce projet est d'illustrer les principes de programmation concurrente et répartie vus en cours. Pour ce faire, nous allons réaliser sur Java une implantation du modèle d'exécution Map-Reduce (MR) similaire à Hadoop proposé par Google.

Le modèle MR permet d'effectuer en parallèle (sur un cluster de machines) des traitements sur un grand volume de données. Les données sont découpées en fragments qui sont répartis (stockés) sur les machines de traitement. Un programme MR est composé d'une procédure map() qui est exécutée en parallèle sur les machines de traitement sur tous les fragments (auxquels on accède donc localement) et produit des résultats qui sont rassemblés sur une machine, et d'une procédure reduce() qui est exécutée (sur cette dernière machine) sur les résultats des map pour obtenir le résultat final.

Ce principe est illustré sur la figure 1.

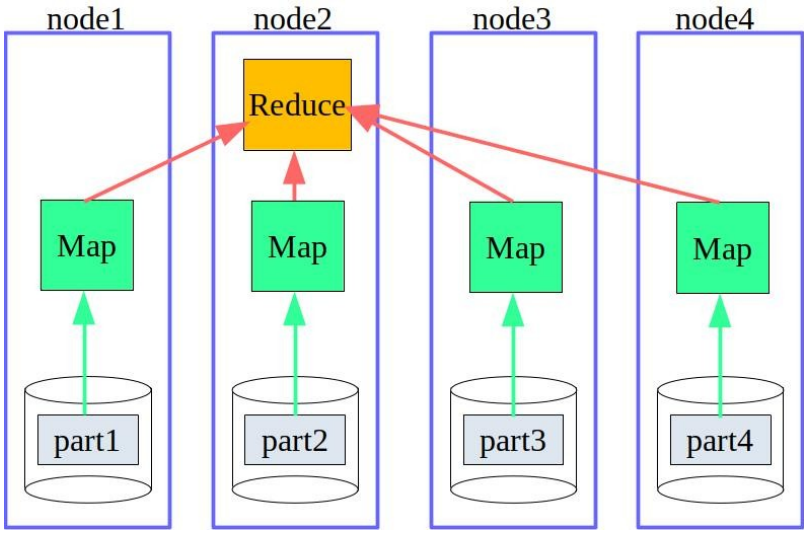


Figure1. Principe MR

L'exemple emblématique utilisé pour illustrer les applications de traitement de grands volumes de données est l'application de comptage de mots (WordCount). En annexes A et B sont donnés les codes de cette application en version MR et en version itérative.

En MR, le fonctionnement de l'application WordCount est illustré sur la figure 2.

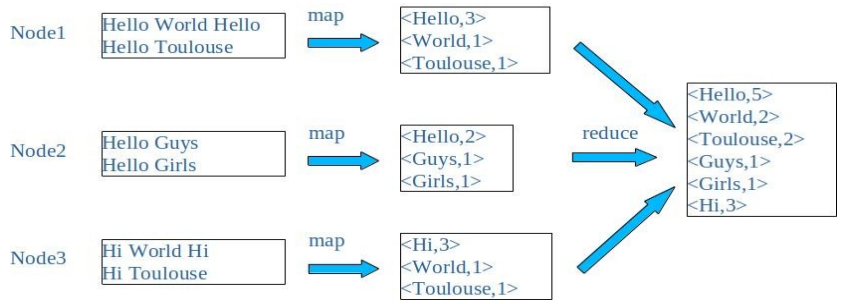


Figure2. Application WordCount

Sur chaque nœud, on a stocké un fragment des données à traiter. La procédure map() est exécutée en parallèle sur chaque nœud. Cette procédure compte les mots dans le fragment et génère un ensemble de KV (key-value pairs), donnant le compte pour chaque mot. Ces KV sont transmis au reduce qui cacule le résultat final sous la forme d'un ensemble de KV. En annexe, l'exemple du WordCount en version MR suit ce schéma et doit être exécutable sur le système Hagidoop que vous devez réaliser.

Le système Hagidoop (global)

Le système Hagidoop est composée de deux services :

- un service HDFS (Hagidoop Distributed File System). Il s'agit d'un système de gestion de fichiers répartis dans lequel un fichier est découpé en fragment, chaque fragment étant stocké sur un des nœuds du cluster.
- un service Hagidoop permettant l'exécution répartie et parallèle des traitements map, la récupération des résultats et l'exécution du reduce.

Le service HDFS

Le service HDFS est représenté sur la figure 3.

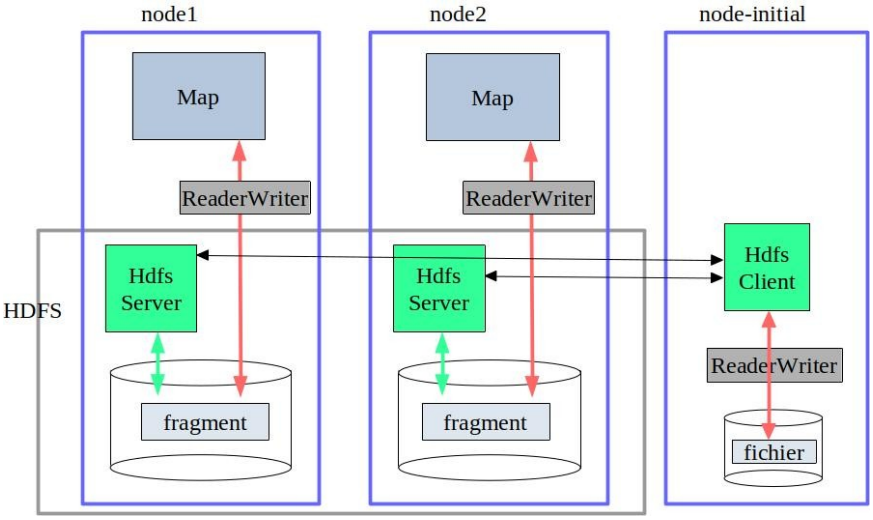


Figure3. Architecture HDFS

HDFS permet de gérer des fichiers fragmentés sur les nœuds. Quand on copie un fichier du file system (FS) local dans le FS HDFS, le fichier (toto.txt) est coupé en fragments qui sont copiés sur les nœuds avec un nom particulier (toto-i.txt). Quand on copie un fichier du FS HDFS dans le FS local, les fragments sont rassemblés pour obtenir le fichier complet sur le FS local (cette dernière fonction ne sera pas utilisée). Les fragments sont de taille variable (en fonction du nombre de nœuds), donc plus on a de nœuds, plus les fragments seront petits pour une taille de fichier donnée.
Les ReaderWriter sont des objets permettant de lire/écrire des KV. Pour HDFS, on lit des KV à partir de fichiers locaux (sur le nœud initial). Ces fichiers locaux peuvent être de deux formats, soit des fichiers texte, soit des fichiers de KV. Les lectures dans ces fichiers sont dites cohérentes, en particulier pour un fichier texte, une lecture lit une ligne sans la couper et la ligne est retournée dans la valeur du KV.
Pour implanter HDFS, on exécute un daemon (HdfsServer) sur chaque nœud, et une classe HdfsClient fournit les commandes de manipulation de HDFS depuis un shell.
Voici un template de HdfsClient :

```
public class HdfsClient {
	public static void HdfsDelete(String fname) {…}
	public static void HdfsWrite(int fmt, String fname) {…}
	public static void HdfsRead(String fname) {…}

	public static void main(String[] args) {
		// java HdfsClient <read|write> <txt|kv> <file>
		// appel des méthodes précédentes depuis la ligne de commande
	}
}
```

La classe HdfsClient peut être utilisée depuis la ligne de commande pour lire, écrire ou détruite des fichiers :
- `HdfsWrite(int fmt, String fname)`
permet d'écrire un fichier dans HDFS. Le fichier fname est lu sur le système de fichiers local, découpé en fragments (autant que le nombre de machines) et les fragments sont envoyés pour stockage sur les différentes machines. fmt est le format du fichier (FMT_TXT ou FMT_KV).
- `HdfsRead(String fname)`
permet de lire un fichier à partir de HDFS. Les fragments du fichier sont lus à partir des différentes machines, concaténés et stockés localement dans un fichier de nom fname.
- `HdfsDelete(String fname)`
permet de supprimer les fragments d'un fichier stocké dans HDFS.

Depuis un shell, on peut par exemple appeler : java HdfsClient write txt toto.txt
Cette appel va lire les lignes du fichier toto.txt (sans les couper) et les stocker dans des fragments sur les différents nœuds.

Nous recommandons d'utiliser les sockets en mode TCP pour implanter la communication entre HdfsClient et HdfsServer. La classe HdfsClient utilise une classe implantant l'interface ReaderWriter (décrite plus loin) pour lire de façon cohérente des KV à partir de fichiers texte ou KV.

Le service Hagidoop
Le service Hagidoop fournit le support pour l'exécution répartie. Un démon (Worker) doit être lancé sur chaque machine. Nous proposons d'utiliser RMI pour la communication entre ce démon et ses clients. L'interface du démon est interne à Hagidoop, donc libre à vous de la définir. Voici toutefois une proposition d'interface du démon :

```
public interface Worker extends Remote {
      public void runMap (Map m, FileReaderWriter reader,
                          NetworkReaderWriter writer)
                          throws RemoteException;
}
```

Les paramètres sont les suivants :
- Map m : il s'agit du programme map à appliquer sur un fragment hébergé sur la machine où s'exécute le démon.
- FileReaderWriter reader : il s'agit d'un objet permettant de lire des KV (dans un format donné par la classe de reader) sur la machine où s'exécute le démon, à partir d'un fichier contenant le fragment sur lequel doit être appliqué le map.
- NetworkReaderWriter writer : il s'agit d'un objet permettant d'écrire des KV sur la machine où s'exécute le démon. Les KV sont envoyés à travers le réseau vers le reduce.

Pour lancer un calcul parallèle depuis le nœud initial, Hagidoop fournit une classe et une méthode :

```
public class JobLauncher {
      public static void startJob (MapReduce mr, int format, String fname) {…}
}
```

Les paramètres sont les suivants :
- MapReduce mr : correspond au programme MR à exécuter en parallèle
- int format : indique le format du fichier en entrée (FMT_TXT ou FMT_KV)
- String fname : le nom du fichier contenant les données à traiter. Ce fichier doit avoir été écrit dans HDFS, ce qui signifie qu'il a été découpé en fragments qui sont répartis sur les machines.

Le comportement de startJob est de lancer des map (avec runMap) sur tous les démons des machines, puis attendre que tous les map soient terminés. Les map génèrent des KV qui sont envoyés au reduce.

Le schéma d'exécution de hagidoop est illustré sur la figure 4.

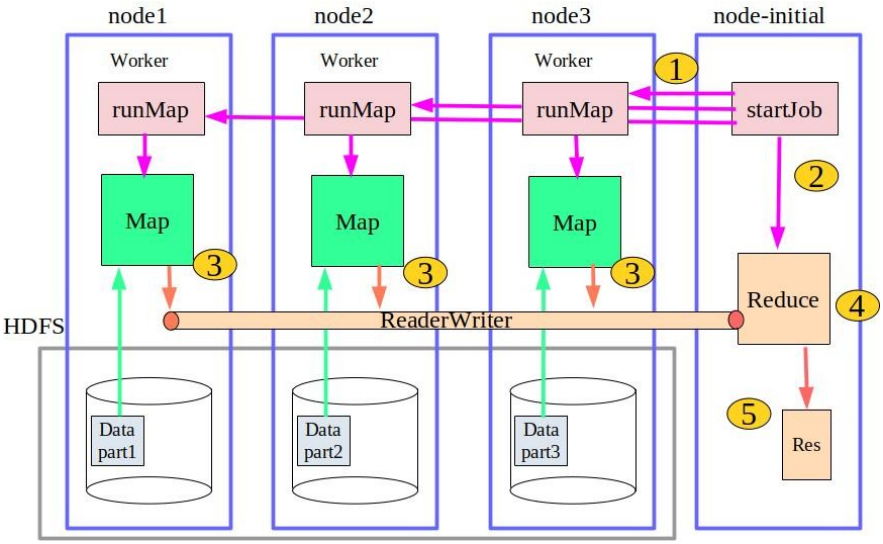


Figure4. Architecture Hagidoop

1. startJob lance les map en appelant runMap sur les Worker (en leur donnant un map, reader et writer).
2. startJob lance le reduce qui récupère les résultats des map (sur une connexion réseau avec les map) et les traite.
3. Les map calculent en lisant localement un fragment (en lisant des KV sur reader, voir interface Map) et génèrent des données (en écrivant des KV sur writer, voir interface Map) envoyées au reduce sur la connexion réseau (l'écriture des KV sur writer envoie les KV sur la connexion).
4. Le reduce traite les données qu'il reçoit (en lisant des KV sur reader, voir interface Reduce) au fur et à mesure, en parallèle des map.
5. Quand tous les map se terminent, le reduce copie le résultat final dans un fichier local (en écrivant des KV sur writer, voir interface Reduce).

La gestion de entrées-sorties

Il faut être en mesure de lire et écrire des données dans des fichiers de formats différents, et dans le fichier initial, le fichier final (résultat) et dans les fragments. Il faut également être en mesurer d'envoyer et recevoir des données sur une connexion réseau entre les Map et le Reducer.

Pour gérer toutes ces communications, vous reposons sur un ensemble de classes et d'interfaces.

Tout d'abord une classe représente la notion de KV.

```
public class KV implements Serializable, Cloneable {
	public static final String SEPARATOR = " - ";
	public String k;
	public String v;
	public KV() {}
	public KV(String k, String v) {
		super();
		this.k = k;
		this.v = v;
	}
	public String toString() {
		return "KV [k=" + k + ", v=" + v + "]";
	}
}
```

La classe KV implémente une paire clé-valeur qui est la façon de représenter les données dans Hagidoop.

Ensuite, un ensemble d'interfaces (Reader, Writer, ReaderWriter) permettent de lire (écrire) des KV à partir de (vers) une origine (destination). Ces origines ou destinations peuvent être des fichiers ou des connexions réseau.

```
public interface Reader {
	public KV read();
}
public interface Writer {
	public void write(KV record);
}
public interface ReaderWriter extends Reader, Writer, Serializable {}
```

Ces interfaces sont spécialisées (FileReaderWriter) pour gérer les entrées-sorties dans des fichiers. En plus de permettre les lectures-écritures de KV, cette interface définit les formats supportés (on peut lire/écrire des KV depuis des fichiers texte et KV), fournit les méthodes pour ouvrir/fermer le fichier et pour savoir où on en est dans le parcours d'un fichier (getIndex()). Les classes implantant cette interface sont utilisées par HDFS pour lire/écrire dans des fichiers (initiaux, finaux et fragments) et par Hagidoop pour lire dans des fragments (Figure 3).

```
public interface FileReaderWriter extends ReaderWriter {
      public static final int FMT_TXT = 0;
      public static final int FMT_KV = 1;
      public void open(String mode);
      public void close();
      public long getIndex();
      public String getFname();
      public void setFname(String fname);
}
```

Ces interfaces sont spécialisées (NetworkReaderWriter) pour gérer les entrées-sorties sur le réseau. En plus de permettre les lectures-écritures de KV, cette interface permet à un Map d'ouvrir une connexion (openClient()) et de fermer une connexion (closeClient()), au Reducer d'ouvrir une connexion (openServer()) et de fermer une connexion (closeServer()) et aussi d'accepter la connexion d'un Map (accept()). Notons que plusieurs Map peuvent se connecter, le Reducer recevant tous les KV écrits par les Map.

```
public interface NetworkReaderWriter extends ReaderWriter {
      public void openServer();
      public void openClient();
      public NetworkReaderWriter accept();
      public void closeServer();
      public void closeClient();
}
```

Le modèle de programmation

Le modèle de programmation MR permet de définir une classe implémentant l'interface MapReduce et fournissant les méthodes map() et reduce().
Une procédure map() peut lire les données (des KV) provenant de son fragment local avec le paramètre reader, et écrire ses résultats (des KV) sur une connexion réseau vers le Reducer avec le paramètre writer.
Une procédure reduce() lit également des KV avec reader (à partir d'une connexion réseau) et écrit des KV sur writer (vers le fichier résultat).

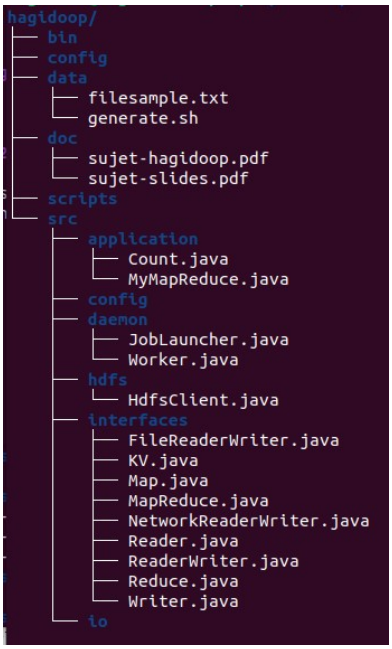
```
public interface Map extends Serializable {
      public void map(Reader reader, Writer writer);
}

public interface Reduce extends Serializable {
      public void reduce(Reader reader, Writer writer);
}

public interface MapReduce extends Map, Reduce {
}
```

En annexe A, vous trouverez l'exemple de l'application WordCount programmé en MR et utilisant ces interfaces.

Proposition d'architecture
Vous n'êtes pas obligés de suivre cette architecture. C'est celle que j'ai choisie et dans laquelle je vous livre les interfaces décrites ci-dessus.



- bin : c'est où mon IDE (eclipse) compile
- config : inclut les fichiers de configuration de hagidoop (notamment un fichier listant les nœuds du cluster)
- data : contient les fichiers de données que l'on peut traiter avec hagidoop. generate.sh est un script permettant de générer de gros fichier pour les évaluations (gros, c'est au moins 1 Gb).
- doc : pour la documentation
- scripts : pour les scripts de déploiement si je veux pouvoir lancer en une commande tous les daemons sur un ensemble de nœuds.
- src/application : les sources des applications. Je donne ici les 2 versions du WordCount en version iterative et MR.
- src/config : les classes qui doivent permettre de configurer hagidoop, par exemple lire le fichier listant les nœuds du cluster
- src/daemon : les classes implantant la partie hagidoop, c'est à dire Worker et JobLauncher
- src/hdfs : les classes implantant HDFS
- src/interface : les interfaces que je vous donne, pour le modèle de programmation MR (Map, Reduce, MapReduce) et pour la gestion des entrées-sorties (KV, Reader, Writer, ReaderWriter, FileReaderWriter, NetworkReaderWriter)
- src/io : les classes implantant les interfaces ci-dessus pour la gestion des entrées-sorties.

Notez que tout ce que je vous donne compile.

Travail à faire
Vous devez implanter le système Hagidoop composé du service Hagidoop et du système de fihcier HDFS.
Votre système doit être facilement utilisable et notamment :
- être configurable par un fichier de configuration qui décrit notamment les machines sur lesquelles Hagidoop s'exécute.
- inclure des scripts de déploiement qui lancent les démons sur les machines et nettoient ces machines.
Vous devez ensuite mener une étude du passage à l'échelle de votre système. Votre système doivent pouvoir être lancé avec plusieurs démons sur la même machine pour profiter de l'exécution parallèle sur des cœurs multiples, et en répartis sur les machines de l'N7.

Annexe A (comptage de mots en MR)

```
public class MyMapReduce implements MapReduce {
	private static final long serialVersionUID = 1L;

	// MapReduce program that compute word counts
	public void map(Reader reader, Writer writer) {

		HashMap<String,Integer> hm = new HashMap<String,Integer>();
		KV kv;
		while ((kv = reader.read()) != null) {
			String tokens[] = kv.v.split(" ");
			for (String tok : tokens) {
				if (hm.containsKey(tok)) hm.put(tok, hm.get(tok)+1);
				else hm.put(tok, 1);
			}
		}
		for (String k : hm.keySet()) writer.write(new KV(k,hm.get(k).toString()));
	}

	public void reduce(Reader reader, Writer writer) {
		HashMap<String,Integer> hm = new HashMap<String,Integer>();
		KV kv;
		while ((kv = reader.read()) != null) {
			if (hm.containsKey(kv.k))
				hm.put(kv.k, hm.get(kv.k)+Integer.parseInt(kv.v));
			else
				hm.put(kv.k, Integer.parseInt(kv.v));
		}
		for (String k : hm.keySet()) writer.write(new KV(k,hm.get(k).toString()));
	}

	public static void main(String args[]) {
		long t1 = System.currentTimeMillis();
		JobLauncher.startJob(new MyMapReduce(), FileReaderWriter.FMT_TXT, args[0]);
		long t2 = System.currentTimeMillis();
		System.out.println("time in ms ="+(t2-t1));
		System.exit(0);
	}
}
```

Annexe B (comptage de mots en itératif)

```
public class Count {

	public static void main(String[] args) {

		try {
			long t1 = System.currentTimeMillis();

			HashMap<String,Integer> hm = new HashMap<String,Integer>();
			LineNumberReader lnr = new LineNumberReader(
						new InputStreamReader(
						new FileInputStream(Project.PATH+"data/"+args[0])));
			while (true) {
				String l = lnr.readLine();
				if (l == null) break;
				String tokens[] = l.split(" ");
				for (String tok : tokens) {
					if (hm.containsKey(tok)) hm.put(tok, hm.get(tok)+1);
					else hm.put(tok, 1);
				}
			}
			BufferedWriter writer = new BufferedWriter(
						new OutputStreamWriter(
						new FileOutputStream("count-res")));
			for (String k : hm.keySet()) {
				writer.write(k+"<->"+hm.get(k).toString());
				writer.newLine();
			}
			writer.close();
			lnr.close();
			long t2 = System.currentTimeMillis();
			System.out.println("time in ms ="+(t2-t1));
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

}
```