Projet Hagidoop
Une implantation du modèle Map-Reduce en Java

Daniel Hagimont

Projet ENSPY, 2023

Objectif du projet

Le but de ce projet est de bien comprendre les principes du traitement de grands volumes de données (big data) vus en cours. Pour ce faire, nous allons réaliser sur Java une implantation similaire à Hadoop (proposé par Google), implantant le modèle d'exécution Map-Reduce (MR).

Cette implantation repose bien naturellement sur les communications socket et sur l'appel à distance RMI, 2 mécanismes de communication fournis par Java.

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). Ces exécutions produisent des résultats qui sont rassemblés sur une machine pour effectuer un reduce.
- d'une procédure reduce() qui est exécutée sur les résultats des map pour obtenir le résultat final.

Présentation du projet

Cette implantation du modèle MR 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 (on suppose qu'il n'y en a qu'un).

La gestion de format

Il faut être en mesure de lire et écrire dans des fichiers de formats différents. Pour chaque format, une classe permet de lire et écrire dans un fichier dans ce format. Ces classes de lecture/écriture de formats sont très importantes, car étant donné que les fichiers peuvent être coupés en fragments par HDFS, il faut qu'HDFS sache faire une coupure cohérente. Par exemple, un fichier de structures ne peut pas être coupé au milieu d'une structure. Dans ce projet, on n'implantera que 2 formats : le format texte et le format KV (Key-Value, voir plus loin).

Une fois lues depuis un fichier et traitées par Hagidoop, les données sont toujours représentées par des paires clé-valeur (Key-Value ou KV).

```
public interface Format extends FormatReader, FormatWriter, Serializable {
        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);
}

public interface FormatReader {
        public KV read();
}

public interface FormatWriter {
        public void write(KV record);
}
```

Une classe implantant un format implante l'interface Format. Nous proposons d'implanter deux classes TxtFormat (permettant de traiter des fichiers textes) et KVFormat (permettant de traiter des fichier de KV).

```
public class KV {
        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. Elle est utilisée à la fois par TxtFormat et KVFormat pour lire ou écrire des données.

Le service HDFS

Le système de fichier est composé d'un démon (HdfsServer) qui doit être lancé sur chaque machine. Une classe HdfsClient permet de manipuler les fichiers dans HDFS. Voici un template de cette classe :

```
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) {
                // appel des méthodes précédentes depuis la ligne de commande
        }
}
```

La classe HdfsClient peut être utilisée depuis la ligne de commande ou directement depuis du code 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 stocké localement dans un fichier de nom fname.

- HdfsDelete(String fname)

permet de supprimer les fragments d'un fichier stocké dans HDFS.

Nous recommandons d'utiliser les sockets en mode TCP pour implanter la communication entre HdfsClient et HdfsServer.

Le service Hagidoop

Le service Hagidoop fournit le support pour l'exécution répartie. Un démon (Daemon) 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 :

```
public interface Daemon extends Remote {
        public void runMap (Map m, Format reader, Format writer, CallBack cb)
                                        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.
- Format reader : il s'agit du fichier (dans un format donné par la classe de reader) sur la machine où s'exécute le démon, contenant le fragment sur lequel doit être appliqué le map.
- Format writer : il s'agit du fichier (dans un format donné par la classe de writer) sur la machine où s'exécute le démon, dans lequel les résultats du map doivent être écrits.
- CallBack cb : il s'agit d'un objet de rappel, appelé lorsque l'exécution du map est terminée

Pour lancer un calcul parallèle, Hagidoop fournit une classe et une méthode :

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

- MapReduce mr : correspond au programme MR à exécuter en parallèle (voir modèle de programmation)
- 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 ont générés des fichiers locaux sur les machines, ces fichiers sont des fragments. On peut alors récupérer le fichier global avec HDFS (HdfsClient.HdfsRead()) , puis appliquer le reduce localement.
Le résultat final est un fichier dont le nom est la concaténation de fname et de "-res".

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 méthode map() peut lire les données (des KV) provenant de son fragment local avec le paramètre reader, et écrire ses résultats (des KV) dans un fragment local avec le paramètre writer.
Une méthode reduce() lit également des KV avec reader et écrit des KV sur writer, mais Hagidoop la fait opèrer sur un fichier unique et local à la machine de lancement du programme (ce fichier est le résultat des map extrait de HDFS).

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

public interface Reduce extends Serializable {
        public void reduce(FormatReader reader, FormatWriter writer);
}

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

Exemples d'application
En annexe A, vous trouverez un exemple d'application programmé en MR. Il s'agit d'une application qui compte les occurrences de chaque mots dans un très grand fichier.
En annexe B, vous trouvez le même programme en itératif.

Travail à faire
Vous devez implanter le système Hagidoop composé du service Hagidoop et du système de fichier 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 réparti sur plusieurs machines sur un cluster que vous aurez configuré comme vu en TP.

Question de recherche

On s’intéresse à des applications qui sont limitées par le débit de chargement des fragments (à partir de HDFS) comme WordCount ou Grep.
Pour ces applications, nous proposons de les rendre plus efficaces en nous appuyant sur la compression de données. Lorsque les fragments sont stockées dans HDFS, ils sont compressés. Et lorsqu’un map lit un fragment, il le décompresse avant d’exécuter la méthode map().
Nous proposons d’utiliser le format de compression LZ4 (trouver une API Java).
Pour expérimenter avec cette approche, nous proposons de suivre les étapes suivantes :
- mesurer le temps de chargement et traitement d’un fragment (en fonction de sa taille), avec ou sans compression.
- mesurer ce même temps lorsque plusieurs chargement/traitement sont exécutés en parallèle sur la même machine (on sature alors encore plus les IO).
- proposer une intégration dans Hagidoop

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(FormatReader reader, FormatWriter writer) {

		HashMap<String,Integer> hm = new HashMap<String,Integer>();
		KV kv;
		while ((kv = reader.read()) != null) {
			StringTokenizer st = new StringTokenizer(kv.v);
			while (st.hasMoreTokens()) {
				String tok = st.nextToken();
				if (hm.containsKey(tok)) hm.put(tok, hm.get(tok).intValue()+1);
				else hm.put(tok, 1);
			}
		}
		for (String k : hm.keySet()) writer.write(new KV(k,hm.get(k).toString()));
	}

	public void reduce(FormatReader reader, FormatWriter 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(), Format.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;
				StringTokenizer st = new StringTokenizer(l);
				while (st.hasMoreTokens()) {
					String tok = st.nextToken();
					if (hm.containsKey(tok))
						hm.put(tok, hm.get(tok).intValue()+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();
		}
	}

}
```