

BigData - Semaine 1

Pierre Nerzic

février-mars 2019

Le cours de cette semaine présente les concepts suivants :

- But du cours
- Mégadonnées
- Système de fichiers distribués
- Programmation « map-reduce » sur Hadoop



Introduction

Pourquoi ce cours ?

Selon [LinkedIn](#), les compétences les plus recherchées depuis plusieurs années sont :

- 1) **Cloud and Distributed Computing** (Hadoop, Big Data)
- 2) Statistical Analysis and Data Mining (R, Data Analysis)
- 10) Storage Systems and Management (SQL)

Voir [cette page](#) pour la liste en France, qui est très similaire.

Préfixes multiplicatifs

Avant de parler de BigData, connaissez-vous les **préfixes** ?

signe	préfixe	facteur	exemple représentatif
k	kilo	10^3	une page de texte
M	méga	10^6	vitesse de transfert par seconde
G	giga	10^9	DVD, clé USB
T	téra	10^{12}	disque dur
P	péta	10^{15}	
E	exa	10^{18}	FaceBook, Amazon
Z	zetta	10^{21}	internet tout entier depuis 2010

Mégadonnées ?

Les **mégadonnées** ou *Big Data* sont des collections d'informations qui auraient été considérées comme gigantesques, impossible à stocker et à traiter, il y a une dizaine d'années.

- **Internet** : Google en 2015 : 10 Eo (10 milliards de Go), **Facebook** en 2018 : 1 Eo de données, 7 Po de nouvelles données par jour, Amazon : 1 Eo.
- **BigScience** : télescopes (1 Po/jour), **CERN** (2 Po lus et écrits/jour, 280 Po de stockage), génome, environnement. . .

NB: ces informations sont très difficiles à trouver.

La raison est que *tout* est enregistré sans discernement, dans l'idée que ça pourra être exploité. Certains prêchent pour que les données collectées soient pertinentes (*smart data*) plutôt que volumineuses.

Distribution données et traitements

Le traitement d'aussi grandes quantités de données impose des méthodes particulières. Un SGBD classique, même haut de gamme, est dans l'incapacité de traiter autant d'informations.

- Répartir les données sur plusieurs machines (jusqu'à plusieurs millions d'ordinateurs) dans des *Data Centers*
 - système de fichiers spécial permettant de ne voir qu'un seul espace pouvant contenir des fichiers gigantesques et/ou très nombreux (HDFS),
 - bases de données spécifiques (HBase, Cassandra, ElasticSearch).
- Traitements du type « map-reduce » :
 - algorithmes faciles à écrire,
 - exécutions faciles à paralléliser.

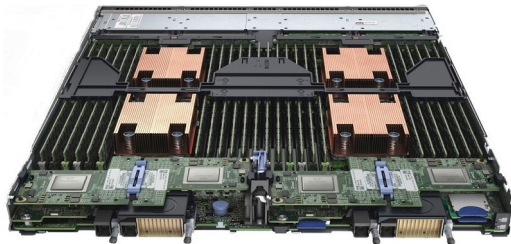
Un Data Center

Imaginez 5000 ordinateurs connectés entre eux formant un *cluster* :



Serveur « lame »

Chacun de ces **PC lames** (*blade computer*) ou *rack server* peut ressembler à ceci (4 CPU multi-cœurs, 1 To de RAM, 24 To de disques rapides, 5000€, prix et technologie en constante évolution) :



Il semble que Google utilise des ordinateurs assez basiques, peu chers mais extrêmement nombreux (10^6), consulter [wikipedia](#).

Machines connectées

Toutes ces machines sont connectées entre elles afin de partager l'espace de stockage et la puissance de calcul.

Le *Cloud* est un exemple d'espace de stockage distribué : des fichiers sont stockés sur différentes machines, généralement en double pour prévenir une panne.

L'exécution des programmes est également distribuée : ils sont exécutés sur une ou plusieurs machines du réseau.

Tout ce module vise à enseigner la programmation d'applications sur un cluster, à l'aide des outils *Hadoop*.

Hadoop ?



Hadoop est un système de gestion de données et de traitements distribués. Il contient de beaucoup de composants, dont :

- HDFS** un système de fichier qui répartit les données sur de nombreuses machines,
- YARN** un mécanisme d'ordonnancement de programmes de type MapReduce.

On va d'abord présenter HDFS puis YARN/MapReduce.

Hadoop File System (HDFS)

Présentation

HDFS est un système de fichiers distribué. C'est à dire :

- les fichiers et dossiers sont organisés en arbre (comme Unix)
- ces fichiers sont stockés sur un grand nombre de machines de manière à rendre invisible la position exacte d'un fichier. L'accès est transparent, quelle que soient les machines qui contiennent les fichiers.
- les fichiers sont copiés en plusieurs exemplaires pour la fiabilité et permettre des accès simultanés multiples

HDFS permet de voir tous les dossiers et fichiers de ces milliers de machines comme un seul arbre, contenant des Po de données, comme s'ils étaient sur le disque dur local.

Organisation des fichiers

Vu de l'utilisateur, HDFS ressemble à un système de fichiers Unix : il y a une racine, des répertoires et des fichiers. Les fichiers ont un propriétaire, un groupe et des droits d'accès comme avec ext4.

Sous la racine /, il y a :

- des répertoires pour les services Hadoop : /hbase, /tmp, /var
- un répertoire pour les fichiers personnels des utilisateurs : /user (attention, ce n'est ni /home, ni /users comme sur d'autres systèmes Unix). Dans ce répertoire, il y a aussi trois dossiers système : /user/hive, /user/history et /user/spark.
- un répertoire pour déposer des fichiers à partager avec tous les utilisateurs : /share

Vous devrez distinguer les fichiers HDFS des fichiers « normaux ».

Commande `hdfs dfs`

La commande `hdfs dfs` et ses options permet de gérer les fichiers et dossiers :

- `hdfs dfs -help`
- `hdfs dfs -ls [noms...]` (pas d'option `-l`)
- `hdfs dfs -cat nom`
- `hdfs dfs -mv ancien nouveau`
- `hdfs dfs -cp ancien nouveau`
- `hdfs dfs -mkdir dossier`
- `hdfs dfs -rm -f -r dossier` (pas d'option `-fr`)

Il faut toutefois noter que les commandes mettent un certain temps à réagir, voir [cette page](#) : ce sont des logiciels écrits en Java avec chargement de très nombreux jars.

D'autre part, nos machines ne sont pas très rapides.

Échanges entre HDFS et le monde

Pour placer un fichier dans HDFS, deux commandes équivalentes :

- `hdfs dfs -copyFromLocal fichiersrc fichierdst`
- `hdfs dfs -put fichiersrc [fichierdst]`

Pour extraire un fichier de HDFS, deux commandes possibles :

- `hdfs dfs -copyToLocal fichiersrc dst`
- `hdfs dfs -get fichiersrc [fichierdst]`

Exemple :

```
hdfs dfs -mkdir -p livres
wget http://www.textfiles.com/etext/FICTION/dracula
hdfs dfs -put dracula livres
hdfs dfs -ls livres
hdfs dfs -get livres/center_earth
```


Comment fonctionne HDFS ?

Comme avec de nombreux systèmes, chaque fichier HDFS est découpé en blocs de taille fixe. Un bloc HDFS = 256Mo (à l'IUT, j'ai réduit à 64Mo). Selon la taille d'un fichier, il lui faudra un certain nombre de blocs. Sur HDFS, le dernier bloc d'un fichier fait la taille restante.

Les blocs d'un même fichier ne sont pas forcément tous sur la même machine. Ils sont copiés chacun sur différentes machines afin d'y accéder simultanément par plusieurs processus. Par défaut, chaque bloc est copié sur 3 machines différentes (c'est configurable).

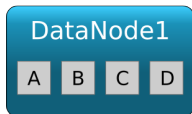
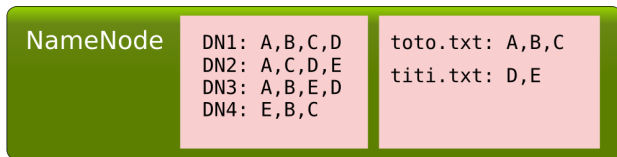
Cette réplication des blocs sur plusieurs machines permet aussi de se prémunir contre les pannes. Chaque fichier se trouve donc en plusieurs exemplaires et à différents endroits.

Organisation des machines pour HDFS

Un cluster HDFS est constitué de machines jouant différents rôles exclusifs entre eux :

- L'une des machines est le maître HDFS, appelé le **namenode**. Cette machine contient tous les noms et blocs des fichiers, comme un gros annuaire téléphonique.
- Une autre machine est le **secondary namenode**, une sorte de namenode de secours, qui enregistre des sauvegardes de l'annuaire à intervalles réguliers.
- Certaines machines sont des **clients**. Ce sont des points d'accès au cluster pour s'y connecter et travailler.
- Toutes les autres machines sont des **datanodes**. Elles stockent les blocs du contenu des fichiers.

Un schéma des nodes HDFS



Les datanodes contiennent des blocs (A, B, C...), le namenode sait où sont les fichiers : quels blocs et sur quels datanodes.

Consulter [cette page](#) pour des explications complètes.

Explications

Les datanodes contiennent des blocs. Les mêmes blocs sont dupliqués (*replication*) sur différents datanodes, en général 3 fois. Cela assure :

- fiabilité des données en cas de panne d'un datanode,
- accès parallèle par différents processus aux mêmes données.

Le namenode sait à la fois :

- sur quels blocs sont contenus les fichiers,
- sur quels datanodes se trouvent les blocs voulus.

On appelle cela les *metadata*.

Inconvénient majeur : panne du namenode = mort de HDFS, c'est pour éviter ça qu'il y a le secondary namenode. Il archive les metadata, par exemple toutes les heures.

Mode *high availability*

Comme le `namenode` est absolument vital pour HDFS mais unique, Hadoop propose une configuration appelée *high availability* dans laquelle il y a 2 autres `namenodes` en secours, capables de prendre le relais instantanément en cas de panne du `namenode` initial.

Les `namenodes` de secours se comportent comme des clones. Ils sont en état d'attente et mis à jour en permanence à l'aide de services appelés *JournalNodes*.

Les `namenodes` de secours font également le même travail que le `secondary namenode`, d'archiver régulièrement l'état des fichiers, donc ils rendent ce dernier inutile.

API Java pour HDFS

API pour utiliser HDFS en Java

Hadoop propose une API Java complète pour accéder aux fichiers de HDFS. Elle repose sur deux classes principales :

- **FileSystem** représente l'arbre des fichiers (*file system*). Cette classe permet de copier des fichiers locaux vers HDFS (et inversement), renommer, créer et supprimer des fichiers et des dossiers
- **FileStatus** gère les informations d'un fichier ou dossier :
 - taille avec `getLen()`,
 - nature avec `isDirectory()` et `isFile()`,

Ces deux classes ont besoin de connaître la configuration du cluster HDFS, à l'aide de la classe **Configuration**. D'autre part, les noms complets des fichiers sont représentés par la classe **Path**

Exemple

Voici quelques manipulations sur un fichier :



```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.Path;

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
Path nomcomplet = new Path("/user/etudiant1", "bonjour.txt");
FileStatus infos = fs.getFileStatus(nomcomplet);
System.out.println(Long.toString(infos.getLen())+" octets");
fs.rename(nomcomplet, new Path("/user/etudiant1","salut.txt"));
```

Dans la suite, `import ...;` correspondra à ces importations.

Informations sur les fichiers

Exemple complet, afficher la liste des blocs d'un fichier :



```
import ...;

public class HDFSinfo {
    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        Path nomcomplet = new Path("apitest.txt");
        FileStatus infos = fs.getFileStatus(nomcomplet);
        BlockLocation[] blocks = fs.getFileBlockLocations(
            infos, 0, infos.getLen());
        for (BlockLocation blocloc: blocks) {
            System.out.println(blocloc.toString());
        }
    }
}
```

Lecture d'un fichier HDFS

Voici un exemple simplifié de lecture d'un fichier texte :



```
import java.io.*;
import ...;
public class HDFSread {
    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        Path nomcomplet = new Path("apitest.txt");
        FSDataInputStream inStream = fs.open(nomcomplet);
        InputStreamReader isr = new InputStreamReader(inStream);
        BufferedReader br = new BufferedReader(isr);
        String line = br.readLine();
        System.out.println(line);
        inStream.close();
        fs.close();
    }
}
```

Création d'un fichier HDFS

Inversement, voici comment créer un fichier :



```
import ...;
public class HDFSwrite {
    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        Path nomcomplet = new Path("apitest.txt");
        if (! fs.exists(nomcomplet)) {
            FSDataOutputStream outputStream = fs.create(nomcomplet);
            outputStream.writeUTF("Bonjour tout le monde !");
            outputStream.close();
        }
        fs.close();
    }
}
```

Compilation et lancement

Compiler et lancer ces programmes avec ce Makefile :



```
HDFSwrite:  HDFSwrite.jar
            hadoop jar HDFSwrite.jar

HDFSread:   HDFSread.jar
            hadoop jar HDFSread.jar

HDFSinfo:   HDFSinfo.jar
            hadoop jar HDFSinfo.jar

%.jar:      %.java
            hadoop com.sun.tools.javac.Main $<
            jar cfe $@ $(basename $<) .
```

Taper `make HDFSwrite` par exemple.

Algorithmes « MapReduce »

Principes

On veut recueillir une information synthétique à partir d'un jeu de données.

Exemples sur une liste d'articles possédant un prix :

- calculer le montant total des ventes d'un article,
- trouver l'article le plus cher,
- calculer le prix moyen des articles.

Pour chacun de ces exemples, le problème peut s'écrire sous la forme de la composition de deux fonctions :

- *map* : extraction/calcul d'une information sur chaque n-uplet,
- *reduce* : regroupement de ces informations.

Exemple

Soient les 4 n-uplets fictifs suivants :

Id	Marque	Modèle	Prix
1	Renault	Clio	4200
2	Fiat	500	8840
3	Peugeot	206	4300
4	Peugeot	306	6140

Calculer le prix maximal, moyen ou total peut s'écrire à l'aide d'algorithmes, étudiés en première année, du type :

pour chaque n-uplet, faire :

 valeur = FonctionM(n-uplet courant)

retourner FonctionR(valeurs rencontrées)

Exemple (suite)

- *FonctionM* est une fonction de correspondance : elle calcule une valeur qui nous intéresse à partir d'un n-uplet,
- *FonctionR* est une fonction de regroupement (agrégation) : maximum, somme, nombre, moyenne, distincts...

Par exemple, *FonctionM* extrait le prix d'une voiture, *FonctionR* calcule le max d'un ensemble de valeurs :

```
tous_les_prix = liste()
pour chaque voiture, faire :
    tous_les_prix.ajouter( getPrix(voiture courante) )
retourner max(tous_les_prix)
```

Pour l'efficacité, les valeurs intermédiaires ne sont pas stockées mais transmises entre les deux fonctions par une sorte de tube (comme dans Unix). Le programme ne s'écrit donc pas tout à fait comme ça.

Exemple en Python

Voici comment on l'écrit en Python2 :



```
data = [  
    {'id':1, 'marque':'Renault', 'modele':'Clio', 'prix':4200},  
    {'id':2, 'marque':'Fiat', 'modele':'500', 'prix':8840},  
    {'id':3, 'marque':'Peugeot', 'modele':'206', 'prix':4300},  
    {'id':4, 'marque':'Peugeot', 'modele':'306', 'prix':6140} ]  
  
# retourne le prix de la voiture passée en paramètre  
def getPrix(voiture): return voiture['prix']  
  
# affiche la liste des prix des voitures  
print map(getPrix, data)  
  
# affiche le plus grand prix  
print reduce(max, map(getPrix, data) )
```

NB: c'est un peu plus lourd en Python3, voir en TP.

Explications

- L'écriture `map(fonction, liste)` applique la fonction à chaque élément de la liste. Elle effectue la boucle « pour » de l'algorithme précédent et retourne la liste des prix des voitures. Ce résultat contient autant de valeurs que dans la liste d'entrée.
- La fonction `reduce(fonction, liste)` agglomère les valeurs de la liste par la fonction et retourne le résultat final¹.

Ces deux fonctions constituent un couple « map-reduce » et le but de ce cours est d'apprendre à les comprendre et les programmer.

Le point clé est la possibilité de paralléliser ces fonctions afin de calculer beaucoup plus vite sur une machine ayant plusieurs cœurs ou sur un ensemble de machines reliées entre elles.

¹En Python, au lieu de `reduce(max, liste)`, on peut écrire `max(liste)` directement.

Parallélisation de Map

La fonction *map* est par nature parallélisable, car les calculs sont indépendants.

Exemple, pour 4 éléments à traiter :

- $\text{valeur}_1 = \text{FonctionM}(\text{element}_1)$
- $\text{valeur}_2 = \text{FonctionM}(\text{element}_2)$
- $\text{valeur}_3 = \text{FonctionM}(\text{element}_3)$
- $\text{valeur}_4 = \text{FonctionM}(\text{element}_4)$

Les quatre calculs peuvent se faire simultanément, par exemple sur 4 machines différentes, à condition que les données y soient copiées.

Remarque : il faut que la fonction mappée soit une pure fonction de son paramètre, qu'elle n'ait pas d'effet de bord tels que modifier une variable globale ou mémoriser ses valeurs précédentes.

Parallélisation de Reduce

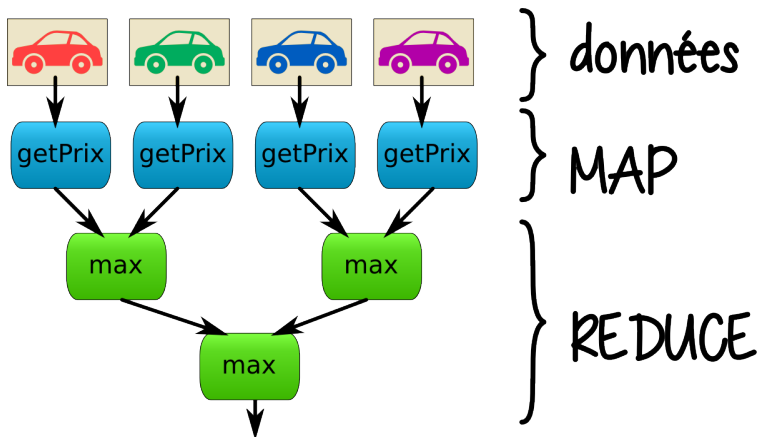
La fonction *reduce* se parallélise partiellement, sous une forme hiérarchique, par exemple :

- $inter_{1 \text{ et } 2} = \text{FonctionR}(\text{valeur}_1, \text{valeur}_2)$
- $inter_{3 \text{ et } 4} = \text{FonctionR}(\text{valeur}_3, \text{valeur}_4)$
- $\text{resultat} = \text{FonctionR}(inter_{1 \text{ et } 2}, inter_{3 \text{ et } 4})$

Seuls les deux premiers calculs peuvent être faits simultanément. Le 3e doit attendre. S'il y avait davantage de valeurs, on procéderait ainsi :

1. calcul parallèle de la FonctionR sur toutes les paires de valeurs issues du map
2. calcul parallèle de la FonctionR sur toutes les paires de valeurs intermédiaires issues de la phase précédente.
3. et ainsi de suite, jusqu'à ce qu'il ne reste qu'une seule valeur.

Un schéma



YARN et MapReduce

Qu'est-ce que YARN ?

YARN (Yet Another Resource Negotiator) est un mécanisme dans Hadoop permettant de gérer des travaux (*jobs*) sur un cluster de machines.

YARN permet aux utilisateurs de lancer des *jobs* MapReduce sur des données présentes dans HDFS, et de suivre (*monitor*) leur avancement, récupérer les messages (*logs*) affichés par les programmes.

Éventuellement YARN peut déplacer un processus d'une machine à l'autre en cas de défaillance ou d'avancement jugé trop lent.

En fait, YARN est transparent pour l'utilisateur. On lance l'exécution d'un programme MapReduce et YARN fait en sorte qu'il soit exécuté le plus rapidement possible.

Qu'est-ce que MapReduce ?

MapReduce est un environnement Java pour écrire des programmes destinés à YARN. Java n'est pas le langage le plus simple pour cela, il y a des packages à importer, des chemins de classes à fournir. . .

Il y a plusieurs points à connaître, c'est la suite de ce cours :

- Principes d'un job MapReduce dans Hadoop,
- Programmation de la fonction Map,
- Programmation de la fonction Reduce,
- Programmation d'un job MapReduce qui appelle les deux fonctions,
- Lancement du job et récupération des résultats.

Commençons d'abord avec le type des données échangées entre Map et Reduce.

Paires clé-valeurs

C'est en fait un peu plus compliqué que ce qui a été expliqué initialement. Les données échangées entre Map et Reduce, et plus encore, dans la totalité du job sont des paires (*clé*, *valeur*) :

- une clé : c'est n'importe quel type de données : entier, texte. . .
- une valeur : c'est n'importe quel type de données

Tout est représenté ainsi. Par exemple :

- un fichier texte est un ensemble de (n° de ligne, ligne).
- un fichier météo est un ensemble de (date et heure, température)

C'est cette notion qui rend les programmes assez étranges au début : les deux fonctions Map et Reduce reçoivent des paires (clé, valeur) et émettent d'autres paires, selon les besoins de l'algorithme.

Map

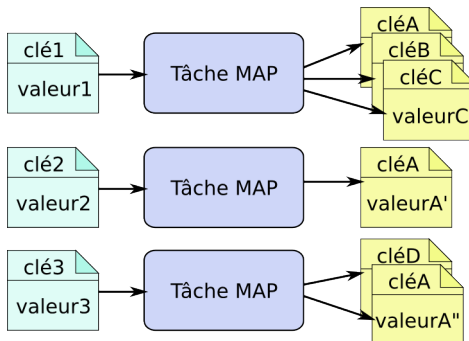
La fonction Map reçoit une paire en entrée et peut produire un nombre quelconque de paires en sortie : aucune, une ou plusieurs, à volonté. Les types des entrées et des sorties sont comme on veut.

Cette spécification très peu contrainte permet de nombreuses choses. En général, les paires que reçoit Map sont constituées ainsi :

- la valeur de type *text* est l'une des lignes ou l'un des n-uplets du fichier à traiter
- la clé de type *integer* est la position de cette ligne dans le fichier (on l'appelle *offset* en bon français)

Il faut comprendre que YARN lance une instance de Map pour chaque ligne de chaque fichier des données à traiter. Chaque instance traite la ligne qu'on lui a attribuée et produit des paires en sortie.

Schéma de Map



Les tâches MAP traitent chacune une paire et produisent 0..n paires.
Il se peut que les mêmes clés et/ou valeurs soient produites.

Reduce

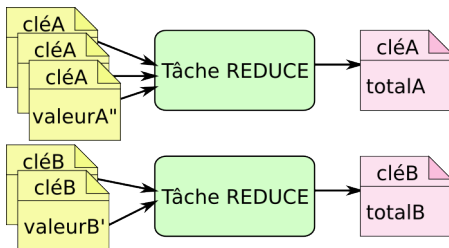
La fonction Reduce reçoit une liste de paires en entrée. Ce sont les paires produites par les instances de Map. Reduce peut produire un nombre quelconque de paires en sortie, mais la plupart du temps, c'est une seule. Par contre, le point crucial, c'est que les paires d'entrée traitées par une instance de Reduce ont toutes la même clé.

YARN lance une instance de Reduce pour chaque clé différente que les instances de Map ont produit, et leur fournit uniquement les paires ayant la même clé. C'est ce qui permet d'agréger les valeurs.

En général, Reduce doit faire un traitement sur les valeurs, comme additionner toutes les valeurs entre elles, ou déterminer la plus grande des valeurs. . .

Quand on conçoit un traitement MapReduce, on doit réfléchir aux clés et valeurs nécessaires pour que ça marche.

Schéma de Reduce



Les tâches Reduce reçoivent une liste de paires ayant toutes la même clé et produisent une paire qui contient le résultat attendu. Cette paire en sortie peut avoir la même clé que celle de l'entrée.

Exemple

Une entreprise de téléphonie veut calculer la durée totale des appels téléphoniques d'un abonné à partir d'un fichier CSV contenant tous les appels de tous les abonnés (n° d'abonné, n° appelé, date, durée d'appel). Ce problème se traite ainsi :

1. En entrée, on a le fichier des appels (1 appel par ligne)
2. YARN lance une instance de la fonction Map par appel
3. Chaque instance de Map reçoit une paire (offset, ligne) et produit une paire (n° abonné, durée) ou rien si c'est pas l'abonné qu'on veut. NB: l'offset ne sert à rien ici.
4. YARN envoie toutes les paires vers une seule instance de Reduce (car il n'y a qu'une seule clé différente)
5. L'instance de Reduce additionne toutes les valeurs des paires qu'elle reçoit et produit une seule paire en sortie (n° abonné, durée totale)

Remarques

En réalité, il n'y a pas qu'une seule instance de Reduce, il y en a plusieurs pour faire la réduction de manière hiérarchique plus rapidement. Car en général l'algorithme qu'on écrit dans la fonction Reduce est une boucle sur chaque valeur reçue.

Également, en réalité, il n'y a pas une instance de Map par ligne de données. C'est la vision qu'on peut avoir en tant que programmeur, mais ça conduirait à un nombre gigantesque d'instances pour traiter un énorme fichier. En fait, YARN instancie un seul « Mappeur » par machine esclave et appelle sa méthode `map` à plusieurs reprises pour traiter les données séquentiellement.

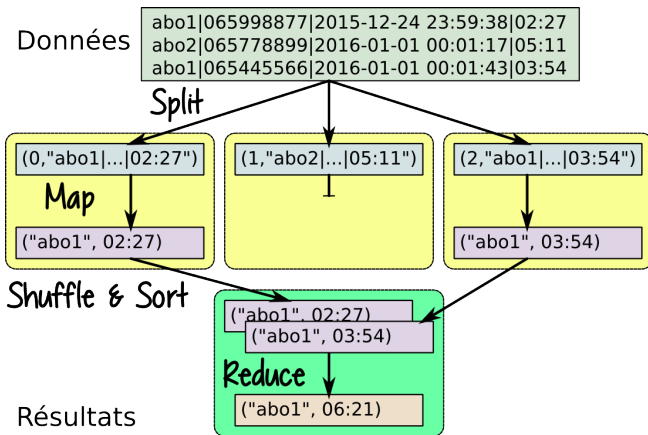
Ce cours fait plusieurs simplifications comme cela afin de rester compréhensible pour une première découverte de Hadoop.

Étapes d'un job MapReduce

Un *job* MapReduce comprend plusieurs phases :

1. Prétraitement des données d'entrée, ex: décompression des fichiers
2. **Split**: séparation des données en blocs traitables séparément et mise sous forme de (clé, valeur), ex: en lignes ou en n-uplets
3. **Map**: application de la fonction map sur toutes les paires (clé, valeur) formées à partir des données d'entrée, cela produit d'autres paires (clé, valeur) en sortie
4. **Shuffle & Sort**: redistribution des données afin que les paires produites par Map ayant les mêmes clés soient sur les mêmes machines
5. **Reduce**: agrégation des paires ayant la même clé pour obtenir le résultat final.

Un schéma



Explication du schéma

1. Au début, YARN se renseigne sur l'emplacement des données auprès du `namenode` et les fait décompresser si besoin par les `datanodes` concernés.
2. La phase *Split* consiste à construire des paires (n° de `n-uplet`, `n-uplet`) à fournir aux tâches *Map*.
3. YARN crée des processus *Map* sur chaque machine contenant une partie des données et leur fournit les paires de leur machine successivement.
4. Chaque tâche *Map* analyse ses données et émet ou non une paire. Ça peut consister à convertir des chaînes en nombres, à faire des calculs, etc.

Explication du schéma (suite)

5. YARN trie les paires sortant de *Map* selon leur clé et les envoie sur la machine qui fait tourner la tâche *Reduce* concernée par cette clé.
6. Les tâches *Reduce* reçoivent une liste de paires et effectuent la réduction des valeurs (*max*, *sum*, *avg*...). Elles émettent seulement la valeur finale. Elles peuvent être mises en cascade quand il y a beaucoup de paires.

API Java pour MapReduce

Présentation

On arrive à la partie la plus technique : la programmation d'un job MapReduce en Java.

Il faut 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 classes génériques (*templates*) paramétrées par les types des clés et des valeurs.

Squelette de Mapper



```
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);
    }
}
```

Explications

La classe Mapper est paramétrée par 4 types. Hélas, 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*. Ça complique légèrement les programmes.

type	description
Text	chaîne UTF8 quelconque
BooleanWritable	représente un booléen
IntWritable	entier 32 bits
LongWritable	entier 64 bits
FloatWritable	réel IEEE 32 bits
DoubleWritable	réel IEEE 64 bits

Types de données MapReduce

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();
```


Interface Writable

Elle permet la *sérialisation*, c'est à dire l'écriture d'une structure de données sous forme d'octets et l'opération inverse, la *désérialisation* qui permet de reconstruire une structure de données à partir d'octets.

La sérialisation est nécessaire pour échanger des données entre machines. Cela fait partie de la technique appelée *Remote Procedure Call* (RPC). 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. . .

Cette interface n'est pas limitée à des types simples mais peut gérer des collections (tableaux, listes, dictionnaires. . .) et classes.

Classe Text

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

Ces méthodes ne sont pas suffisantes. Il faudra souvent convertir les Text en chaînes.

Squelette de Reducer



```
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);
    }
}
```


Explications

La méthode `reduce` reçoit une collection de valeurs venant du *Mapper*. `CléI` 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`...

Une chose cruciale n'est pas du tout vérifiée par Java : il est obligatoire que les types des clés `TypCléI` 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 passera à la compilation mais plantera à l'exécution.

Squelette de TraitementDriver

Voici la classe principale qui crée et lance le job MapReduce : 

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

    public int run(String[] args) throws Exception
    {
        /* voir transparent suivant */
    }
}
```

Squelette de TraitementDriver (cœur)

La méthode run contient ceci :



```
public int run(String[] args) throws Exception
{
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "traitement");
    job.setJarByClass(TraitementDriver.class);

    job.setMapperClass(TraitementMapper.class);
    job.setReducerClass(TraitementReducer.class);

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

    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
}
```

Explications

La méthode `run` est chargée de créer et lancer un Job. Il faut noter que la spécification Hadoop a beaucoup changé depuis les premières versions. Il faut actuellement faire ainsi :

1. Obtenir une instance de `Configuration`. Elle contient les options telles que les formats des fichiers, leur nom HDFS complet, leur *codec* de compression... voir le prochain cours.
2. Créer un Job, lui indiquer les classes concernées : *mapper* et *reducer*.
3. Fournir les noms complets des fichiers à traiter et à produire.
4. Indiquer les types des clés et valeurs. Par défaut, ce sont des `Text`.
5. Attendre la fin du job et retourner un code d'erreur.

Davantage de détails au prochain cours.

Compilation et lancement d'un traitement

1. Compilation

```
hadoop com.sun.tools.javac.Main Traitement*.java
```

2. Emballage dans un fichier jar. NB: c'est plus compliqué quand il y a des packages.

```
jar cfe Traitement.jar TraitementDriver Traitement*.class
```

3. Préparation : mettre en place les fichiers à traiter, supprimer le dossier de sortie

```
hdfs dfs -rm -r -f sortie
```

4. Lancement

```
yarn jar Traitement.jar entree sortie
```

5. Résultats dans le dossier sortie

```
hdfs dfs -cat sortie/part-r-00000
```